4

Shortest Paths and Dynamic Programming

The shortest path problem is a classical and important combinatorial problem that arises in many contexts. We are given a directed graph and a cost or "length" a_{ij} for each arc (i,j). The length of a path $(i, i_1, i_2, \ldots, i_k, 1)$ from node i to node 1 with arcs $(i, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, 1)$ is defined to be the sum of the arc lengths $\left(a_{ii_1} + a_{i_1 i_2} + \cdots + a_{i_{k-1} i_k} + a_{i_k 1}\right)$. The problem is to find a path of minimum length (or shortest path) from each node i to node 1.

There are many applications of the shortest path problem. One example, of particular relevance to distributed computation, arises in the context of routing data within a computer communication network. Here the length a_{ij} represents a measure of cost (such as average delay) for crossing link (i,j). Thus, a shortest path is a minimum cost path and can be viewed as a desirable path for routing data. An interesting feature here is that the communication network defines the graph of the shortest path problem, and each node is a processor that can participate in the numerical solution.

Shortest path applications arise also in other types of routing problems, involving, for example, the flow of vehicles, materials, etc. Other examples include problems of heuristic search, and deterministic optimal control problems, where the trajectory of a dynamic system is to be optimized over a given time interval [Ber87]. Here the number of nodes is often very large, and parallel computation may be required to reduce the computation time to an acceptable level.

Finally, the shortest path problem frequently arises as a subroutine in algorithms for solving other, more complicated problems. In this context, the shortest path subroutine may have to be called many times, so its fast parallel execution can be critical for the success of the overall algorithm.

The shortest path problem formulation is a special case of a more general modeling technique known as dynamic programming, which deals with the issue of making optimal decisions sequentially. Each decision results in a cost, but also affects the options of subsequent decisions, so the objective is to strike a balance between incurring a low cost for the present decision and avoiding future situations where high costs are inevitable. We can also view the shortest path problem as a problem of sequential decision making. Starting at node i, the first decision is to select an incident arc (i, i_1) of node i, then to select an arc (i_1, i_2), and so on until the destination node is reached through a final arc $(i_k, 1)$. In the first decision one must balance the desire to select an arc (i, j) with small length a_{ij} with the desire to avoid going to a node j that is "far" from the destination. This tradeoff is captured in the equation

$$x_i^* = \min_j (a_{ij} + x_j^*), \qquad i = 2, \ldots, n,$$

$$x_1^* = 0,$$

which, as we shall see, is satisfied by the shortest path lengths x_i^*, $i = 1, \ldots, n$. The dynamic programming algorithm for the shortest path problem has the form

$$x_i := \min_j (a_{ij} + x_j), \qquad i = 2, \ldots, n,$$

$$x_1 := 0,$$

which is reminiscent of the relaxation methods of Chapters 2 and 3. This algorithm is particularly well suited for parallel or distributed implementation since the minimization over j in the previous equation can be carried out in parallel for all nodes $i \neq 1$. Its convergence to the shortest path lengths will be shown in Subsection 4.1.1 for a broad range of initial conditions. In Subsection 4.1.2, we will analyze the complexity of other shortest path methods that are well suited for distributed implementation, and we will compare them with the preceding algorithm.

In this chapter, we will also discuss dynamic programming problems that are more general than the shortest path problem in that, once a decision is selected at a given node, the next node is not predictable, but rather is chosen according to a known probability distribution that depends on the selected decision. This leads to a model involving a finite–state Markov chain, the transition probabilities of which are influenced by the choice of decision. We first consider this model in Section 4.2 for the simpler case where there are no decisions to be made or, equivalently, the transition probabilities are independent of the decision. We then consider the effect of decisions in Section 4.3. Much of our dynamic programming analysis is based on a monotonicity property of the mapping underlying the dynamic programming algorithm [Prop. 3.1(a) in Section

4.3]. Under additional conditions, this mapping is also a contraction with respect to a maximum norm [Prop. 3.1(c) and Exercise 3.3]. We use these two properties to show convergence of the algorithm to the correct solution. In Chapter 6, we show that these properties also guarantee convergence in a totally asynchronous distributed environment.

4.1 THE SHORTEST PATH PROBLEM

The shortest path problem is defined in terms of a directed graph consisting of n nodes, which are numbered $1, \ldots, n$. We denote by $A(i)$ the set of all nodes j for which there is an outgoing arc (i, j) from node i. Node 1 is a special node called the *destination*. We assume that $A(1)$ is empty, that is, the destination has no outgoing arcs. We are given a scalar a_{ij} for each arc (i, j), which we call the length of (i, j). We define the *length* of a path $\{(i, i_1), (i_1, i_2), \ldots, (i_k, j)\}$ starting from node i and ending at node j to be the sum of the lengths of its arcs $(a_{ii_1} + a_{i_1 i_2} + \cdots + a_{i_k j})$. The problem is to find a path of minimum length (or shortest path) from each node i to the destination. Note here that we are optimizing over paths consisting exclusively of forward arcs (such paths are called positive; see Appendix B). When we refer to a path or a cycle in connection with the shortest path problem, we implicitly assume that it is positive. We assume the following:

Assumption 1.1. (*Connectivity*) There exists a path from every node $i = 2, \ldots, n$ to the destination node 1.

Assumption 1.2. (*Positive Cycle*) Every cycle has positive length.

We will show that the shortest path lengths x_i^*, $i = 1, \ldots, n$, also called *shortest distances*, are the unique solution of the system

$$x_i^* = \min_{j \in A(i)} (a_{ij} + x_j^*), \qquad i = 2, \ldots, n, \tag{1.1a}$$

$$x_1^* = 0 \tag{1.1b}$$

(known as *Bellman's equation*). Furthermore, we will show that the iteration

$$x_i := \min_{j \in A(i)} (a_{ij} + x_j), \qquad i = 2, \ldots, n, \tag{1.2a}$$

$$x_1 := 0 \tag{1.2b}$$

(also known as the *Bellman–Ford algorithm*) converges to this solution for an arbitrary initial vector x with $x_1 = 0$.

Note that the Bellman–Ford algorithm is particularly well suited for parallel and distributed implementations since the iteration for each node i can be carried out simultaneously with the iteration for every other node. The version of the Bellman–Ford

algorithm that we will focus on in this section can be viewed as a Jacobi relaxation method for solving the system of n nonlinear equations with n unknowns specified by Bellman's equation (1.1). There is also a Gauss–Seidel version, which is considered in Exercise 1.2, and can be viewed as a coordinate ascent method in the context of a dual network optimization problem (see Subsection 5.2.1). A totally asynchronous implementation of the Bellman–Ford algorithm will be discussed in Section 6.4.

One possible set of initial conditions in the Bellman–Ford algorithm is $x_i = \infty$ for $i \neq 1$ and $x_1 = 0$; this is the choice most often discussed in the literature. Indeed, in the absence of additional information, this is a good choice, and results in polynomially bounded running time for the algorithm, as will be shown shortly. Our interest in arbitrary initial conditions stems from certain applications where the shortest path problem must be solved repeatedly and in real time, as the arc lengths change by small increments. A small change in the arc lengths implies a small change in the shortest path lengths, so it may be advantageous in terms of speed of convergence to restart the Bellman–Ford algorithm using as initial conditions the previous shortest path lengths (or approximations thereof). Another advantage of this approach in a distributed implementation is that it does not require a potentially complex and time–consuming restart/resynchronization procedure to inform all the processors of the change in problem data and to restart the algorithm with a predetermined set of initial conditions. In this context, the processors of the distributed system can simply incorporate the changes of the arc lengths in their iterations at the time that they become aware of them. In other words, the processors keep on executing their portion of the iteration (1.2) using the latest information available regarding the values of the arc lengths. Naturally, in order for this scheme to be workable, the arc lengths should not change too frequently relative to the speed of convergence of the Bellman–Ford algorithm. For further discussion of this type of algorithm in the context of routing data in a communication network, we refer the reader to Subsection 5.2.4 of [BeG87].

4.1.1 The Bellman–Ford Algorithm

The kth iteration of the Bellman–Ford algorithm has the form

$$x_i^k = \min_{j \in A(i)} (a_{ij} + x_j^{k-1}), \qquad i = 2, \ldots, n, \tag{1.3a}$$

$$x_1^k = 0. \tag{1.3b}$$

Regarding initial conditions, we assume that $x_1^0 = 0$, and that for $i = 2, \ldots, n$, x_i^0 is either a real number or $+\infty$. We say that the algorithm *terminates after k iterations* if $x_i^k = x_i^{k-1}$ for all i.

Given nodes $i \neq 1$ and $j \neq 1$, we define

$$\begin{aligned} w_{ij}^k &= \text{minimum path length over all paths from } i \text{ to } j, \\ &\quad \text{and having } k \text{ arcs } (w_{ij}^k = \infty \text{ if there is no such path}). \end{aligned} \tag{1.4}$$

To complete the definition of w_{ij}^k, we define for all $i \neq 1$

$$w_{i1}^k = \text{minimum path length over all paths from } i \text{ to } 1 \text{ having } k \text{ arcs or less} \quad (w_{i1}^k = \infty \text{ if there is no such path}). \tag{1.5}$$

Note that the path of minimum length in the definition of w_{ij}^k can contain cycles; this will certainly happen if $j \neq 1$ and $k \geq n-1$.

The following lemma is very useful:

Lemma 1.1. There holds

$$x_i^k = \min_{j=1,\ldots,n} (w_{ij}^k + x_j^0), \qquad \forall\, i = 2,\ldots,n, \text{ and } k \geq 1. \tag{1.6}$$

Proof. We use induction. By the definitions of x_i^1 and w_{ij}^1 [Eqs. (1.3)–(1.5)], the result holds for $k = 1$. Assume that it holds for some $k \geq 1$. For any nodes $i \neq 1$ and j, let us denote by P_{ij}^k the set of paths appearing in the definition of w_{ij}^k [Eqs. (1.4)–(1.5)]. We have, using the convention $a_{i1} = \infty$ if $1 \notin A(i)$,

$$\begin{aligned}
\min_{j=1,\ldots,n} \left(w_{ij}^{k+1} + x_j^0\right) &= \min_{\substack{(i,i_1,\ldots,i_m,j)\in P_{ij}^{k+1} \\ j=1,\ldots,n}} \left(a_{ii_1} + a_{i_1i_2} + \cdots + a_{i_mj} + x_j^0\right) \\
&= \min\left(a_{i1},\ \min_{i_1\in A(i), i_1\neq 1}\left(a_{ii_1} + \min_{\substack{(i_1,i_2,\ldots,i_m,j)\in P_{i_1j}^k \\ j=1,\ldots,n}} \left(a_{i_1i_2} + \cdots + a_{i_mj} + x_j^0\right)\right)\right) \\
&= \min\left(a_{i1},\ \min_{i_1\in A(i), i_1\neq 1}\left(a_{ii_1} + \min_{j=1,\ldots,n}\left(w_{i_1j}^k + x_j^0\right)\right)\right) \\
&= \min_{i_1\in A(i)} \left(a_{ii_1} + x_{i_1}^k\right) \\
&= x_i^{k+1},
\end{aligned} \tag{1.7}$$

where we used the induction hypothesis to establish the next to last equality. The induction is complete. **Q.E.D.**

The preceding lemma yields the following result:

Proposition 1.1. Let the Connectivity and Positive Cycle Assumptions 1.1 and 1.2 hold:

(a) There exists a shortest path from every node $i \neq 1$ to node 1. Furthermore, every one of these shortest paths has at most $n - 1$ arcs.

(b) For any set of initial conditions, the Bellman–Ford algorithm terminates after some finite number k of iterations, with x_i^k equal to the shortest distances x_i^*, $i = 1, \ldots, n$.

(c) If $x_i^0 \geq x_i^*$ for all $i \neq 1$, then the Bellman–Ford algorithm yields the shortest distances in at most m^* iterations, and terminates after at most $m^* + 1$ iterations, where

$$m^* = \max_{i=2,\ldots,n} m_i \leq n - 1, \tag{1.8}$$

and m_i is the smallest number of arcs contained in a shortest path from i to 1.

(d) The shortest distances x_i^*, $i = 1, \ldots, n$, are the unique solution of Bellman's equation (1.1).

Proof.

(a) A path from a node $i \neq 1$ to node 1 containing more than $n - 1$ arcs must contain one or more cycles, which, by the Positive Cycle Assumption 1.2, have positive length. By deleting the cycles from the path, we can obtain a shorter path with no more than $n - 1$ arcs. Therefore, only paths with $n - 1$ or less arcs are candidates for optimality. By the Connectivity Assumption 1.1, there exists at least one path having $n - 1$ or less arcs, and there is a finite number of such paths. Therefore, there must exist a shortest path.

(b) Since all cycles have positive length, we have for all $i \neq 1$ and $j \neq 1$, $w_{ij}^k \to \infty$ as $k \to \infty$, and $w_{i1}^k = x_i^* < \infty$ for all $k \geq n-1$. From Lemma 1.1 [cf. Eq. (1.6)], it follows that $x_i^k = x_i^*$ for all sufficiently large k.

(c) Consider first the initial conditions $x_i^0 = \infty$, for all $i \neq 1$. Then from Lemma 1.1 we see that, for each $i \neq 1$ and k, x_i^k is the shortest distance from i to 1 using paths with k arcs or less. Hence $x_i^k = x_i^*$ for all i and $k \geq m^*$. Consider next any set of initial conditions with $x_i^* \leq x_i^0$ for all $i \neq 1$. From part (b) we have that the shortest distances x_i^* solve Bellman's equations. Therefore

$$x_i^* = \min_{j \in A(i)} (a_{ij} + x_j^*) \leq \min_{j \in A(i)} (a_{ij} + x_j^0) = x_i^1, \qquad \forall\, i = 2, \ldots, n,$$

and by repeating this argument, we obtain

$$x_i^* \leq x_i^k, \qquad \forall\, i \text{ and } k.$$

Let $\tilde{x}_i^k$ be the iterates of the Bellman–Ford algorithm, starting from the initial conditions $\tilde{x}_i^0 = \infty$, $i = 2, \ldots, n$. We have, using a similar argument as before,

$$x_i^* \leq x_i^k \leq \tilde{x}_i^k, \qquad \forall\, i \text{ and } k.$$

Since, as shown earlier, we have $\tilde{x}_i^k = x_i^*$ for all i and $k \geq m^*$, the desired conclusion follows.

(d) If we start the Bellman–Ford algorithm with a solution of Bellman's equation, we terminate after a single iteration, so by part (b), this solution must equal the shortest distances. **Q.E.D.**

Figure 4.1.1 gives an example showing how Prop. 1.1 fails if the cycle lengths are assumed nonnegative instead of positive.

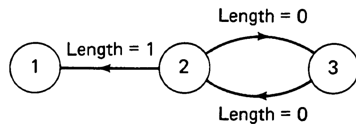

Figure 4.1.1 Shortest path problem involving a cycle of zero length. Here the shortest distances are $x_1^* = 0$, $x_2^* = x_3^* = 1$ and satisfy Bellman's equation. The zero vector also satisfies Bellman's equation, and if the Bellman–Ford algorithm is started with that vector, it will make no progress toward the shortest distance vector.

When the Bellman–Ford algorithm does not terminate, and the Connectivity Assumption 1.1 is known to hold, it follows from Prop. 1.1 that the Positive Cycle Assumption 1.2 is violated, and there exists a cycle with nonpositive length. If all cycle lengths are assumed nonnegative (rather than positive), then it is clear that there exist shortest paths with $n-1$ arcs or less from every $i \neq 1$ to the destination. Lemma 1.1 then implies that the algorithm finds the shortest distances in $n-1$ steps or less when the initial conditions

$$x_i^0 = \infty, \qquad \forall\, i \neq 1 \tag{1.9}$$

are used. If there is a cycle of negative length, then for the same initial conditions, we will have $x_i^n < x_i^{n-1}$ for some $i \neq 1$ (Exercise 1.1), and this can be used to detect the presence of a negative length cycle.

Generally, the number of iterations for termination depends strongly on the initial conditions. This number is $m^* + 1$ when the initial conditions $x_i^0 = \infty$, $i \neq 1$, are used, as shown in Prop. 1.1. For other initial conditions, this number can be much larger, and may depend on the size of the arc lengths (see Fig. 4.1.2). It appears plausible that the number of iterations will often be smaller than m^* if the initial conditions x_i^0 are chosen to be close to their eventual final values x_i^*. A typical situation occurs when the initial conditions are the shortest distances corresponding to arc lengths that either differ slightly from the arc lengths of the problem at hand or else are the same except for a few relatively inconsequential arcs. The following analysis is geared toward estimating the number of iterations for initial conditions of this type. Simple examples show that even if a single arc length changes by a small amount, it is possible that the number of iterations required is as large as $n - 1$. In special cases, however, the number is much smaller (see e.g. Exercise 1.4). The following proposition estimates the number of iterations in terms of the scalars

$$\beta = \max_{i=2,\ldots,n} (x_i^* - x_i^0), \tag{1.10}$$

$$L = \min_{\text{All cycles}} \frac{\text{Length of the cycle}}{\text{Number of arcs on the cycle}}. \tag{1.11}$$

The scalar L of Eq. (1.11) is defined only for graphs that contain at least one cycle, and is known as the *minimum cycle mean.* Algorithms for computing L are considered in [Law67] and [Kar78].

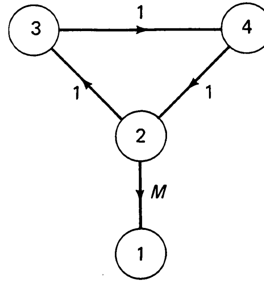

Figure 4.1.2 Example shortest path problem where the number of iterations of the Bellman–Ford algorithm depends on the size of the arc lengths. Here arcs (2,3), (3,4), and (4,2) have length 1, and arc (2,1) has a large length M. If initially $x_i^0 = 0$ for all i then, for all $k \leq M$, after k iterations we have

$$x_i^k = k, \qquad i = 2, 3, 4,$$

$$x_1^k = 0,$$

so the algorithm terminates after $M + 3$ iterations. By contrast, $m^* = 3$, and only 4 iterations are needed for the initial condition $x_i^0 = \infty$, $i = 2, 3, 4$. Similarly, the number of iterations is $\Theta(M)$ when the initial conditions are the shortest distances corresponding to unity length for all arcs. Therefore a large, length–dependent number of iterations may be necessary to recompute the shortest distances following a length increase of a single arc lying on a shortest path.

Proposition 1.2. Let the Connectivity and Positive Cycle Assumptions 1.1 and 1.2 hold. The Bellman–Ford algorithm terminates after at most $\bar{k} + 1$ iterations ($x_i^k = x_i^*$ for all i and $k \geq \bar{k}$), where

$$\bar{k} = \begin{cases} m^*, & \text{if } \beta \leq 0, \\ n - 1, & \text{if } \beta > 0 \text{ and the graph is acyclic,} \\ n - 2 + \lceil \beta / L \rceil, & \text{if } \beta > 0 \text{ and the graph has cycles,} \end{cases} \tag{1.12}$$

m^* is given by

$$m^* = \max_{i=2,\ldots,n} m_i \leq n - 1, \tag{1.13}$$

and m_i is the smallest number of arcs contained in a shortest path from i to 1.

Proof. If $\beta \leq 0$, then $\bar{k} = m^*$ and the result has already been proved in Prop. 1.1(c). Assume $\beta > 0$ and suppose that, for some $k \geq m^*$, the algorithm has not found the shortest distances after k iterations, i.e., $x_i^* \neq x_i^k$ for some i. Since $w_{i1}^k = x_i^*$ for $k \geq m^*$, by Lemma 1.1 we must have $x_i^* \geq x_i^k$ for all i, and furthermore, for some i, and $j \neq 1$, we must have

$$x_i^* > x_i^k = w_{ij}^k + x_j^0.$$

Consider a minimum length path from i to j involved in the definition of w_{ij}^k. This path has k arcs, it does not pass through node 1, and either it contains no cycles, or else it can be decomposed into a simple path from i to j with length L_{ij}, and a nonempty collection of cycles with total length $w_{ij}^k - L_{ij}$ (see the Path Decomposition Theorem of Appendix B). In the former case we obtain $k < n - 1$. In the latter case we argue as follows: since the number of arcs in the simple path is no more than $n - 2$, the number of arcs in the cycles is no less than $k - (n - 2)$, and therefore

$$w_{ij}^k - L_{ij} \geq \big[k - (n-2)\big]L,$$

where L is given by Eq. (1.11). It follows that

$$x_i^* > x_i^k = w_{ij}^k + x_j^0 \geq L_{ij} + x_j^0 + \big[k - (n-2)\big]L.$$

Using the relations $x_j^0 \geq x_j^* - \beta$ and $L_{ij} + x_j^* \geq x_i^*$, we obtain

$$x_i^* > L_{ij} + x_j^* - \beta + \big[k - (n-2)\big]L \geq x_i^* - \beta + \big[k - (n-2)\big]L.$$

Therefore, $k < \max\{n - 1, n - 2 + \lceil \beta/L \rceil\} = n - 2 + \lceil \beta/L \rceil$. Thus we have shown that if $\beta > 0$ and the algorithm has not found the shortest distances at iteration k, then either $k < n - 1$ (if the graph is acyclic) or $k < n - 2 + \lceil \beta/L \rceil$ (if the graph has cycles). This completes the proof. **Q.E.D.**

Figure 4.1.3 shows that the estimate of Prop. 1.2 on the number of iterations is tight in the case of a graph with cycles. To show that the estimate is tight for acyclic graphs, consider the graph with arcs $(i+1, i)$, $i = 1, \ldots, n-1$, and $(n, 1)$. Let $a_{(i+1)i} = 1$, let $a_{n1} = n - (3/2)$, and consider the algorithm with zero initial conditions. A straightforward calculation shows that the algorithm finds the shortest distances after $n - 1$ iterations while $m^* = n - 2$.

From Prop. 1.2, it is seen that the number of iterations is guaranteed to be relatively small if the initial conditions x_i^0 are not much smaller than the true shortest distances x_i^*. If x_i^0 is much smaller than x_i^* for some i, we may ignore the given initial conditions, and start the algorithm from the infinite initial conditions of Eq. (1.9), thereby guaranteeing termination in $m^* + 1$ iterations. A related procedure which changes selectively some of the initial conditions is given in Exercise 1.4.

Timing Analysis of the Bellman–Ford Algorithm

It can be seen that each iteration of the Bellman–Ford algorithm involves $O(|A|)$ additions and comparisons, where $|A|$ is the number of arcs. Hence, for the initial conditions $x_i^0 = \infty$, $i \neq 1$, the serial solution time for the problem is $O\big(m^*|A|\big)$. For other initial conditions, the solution time is $O\big(\bar{k}|A|\big)$ where $\bar{k}$ is the estimate on the number of iterations given by Eq. (1.12).

We consider now two types of synchronous parallel implementations of the Bellman–Ford algorithm. In the first type, we have a distributed system involving an interconnection network of n processors that is identical with the graph of the shortest path

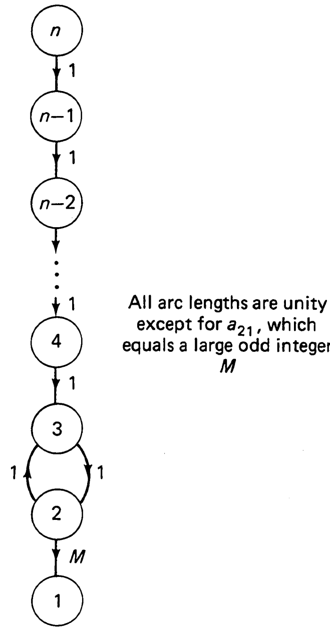

Figure 4.1.3 Example shortest path problem showing that the estimate of Prop. 1.2 on the number of iterations of the Bellman–Ford algorithm is tight. Here all arcs have unity length except for (2,1) which has length equal to the positive *odd* integer M. The initial conditions are

$$x_i^0 = \max\{0, i-2\}, \qquad i = 1, \dots, n.$$

The generated sequence of iterates for nodes 2 and 3 are

$$\{x_2^k\} = \{0, 2, 2, 4, 4, \dots, 2\lfloor M/2 \rfloor, 2\lfloor M/2 \rfloor, M, M, \dots\},$$

$$\{x_3^k\} = \{0, 1, 3, 3, 5, \dots, 2\lfloor M/2 \rfloor - 1, 2\lfloor M/2 \rfloor + 1, 2\lfloor M/2 \rfloor + 1, M+1, \dots\}.$$

For $i \geq 2$ the shortest distance $x_i^* = i - 2 + M$ is found in $i - 2 + M$ iterations. At iteration $n - 2 + M$ all shortest distances have been obtained. In this problem $\beta = M$, $L = 1$, so the estimate of Prop. 1.2 is tight.

problem in the sense that for every arc (i, j) of the latter problem, there is a bidirectional communication link connecting processors i and j. An iteration consists of the update

$$x_i := \min_{j \in A(i)} (a_{ij} + x_j), \tag{1.14}$$

at each processor $i \neq 1$, followed by a transmission of the result to the neighbor processors j with $i \in A(j)$. It is clear then that the time per iteration is $O(r)$, where r is the maximum number of outgoing arcs from a node in the graph, that is,

$$r = \max_{i \neq 1} |A(i)|. \tag{1.15}$$

Note that if each processor itself consists of a parallel computing system with r processors, then the time per iteration becomes $O(\log r)$, assuming negligible communication delays. It is natural to synchronize this implementation using the local synchronization method, whereby a processor can proceed to the next iteration once it receives the results of the previous iteration from its neighboring processors. It is still necessary for each processor to send its shortest distance estimate to all its neighbors at the end of every iteration, even if this estimate has not changed over the previous iteration. This can be wasteful both in terms of time and in terms of communication resources. In Section

6.4, we will see that it is advantageous in this respect to use an asynchronous version of the Bellman–Ford algorithm, where shortest distance estimates are transmitted only when they change values, and processors do not have to wait for messages from all their neighbors before updating their estimate of shortest distance.

The second type of synchronous parallel implementation involves the use of a regular interconnection network of processors such as a hypercube. The situation here is quite similar as for the matrix–vector type of calculations discussed in Subsection 1.3.6. This becomes evident when we compare the term

$$\min_{j \in A(i)} (a_{ij} + x_j) \tag{1.16}$$

in the Bellman–Ford iteration, with the ith coordinate of the product Ax

$$[Ax]_i = \sum_{j \in A(i)} a_{ij} x_j, \tag{1.17}$$

where A is the matrix with entries a_{ij} for $j \in A(i)$ and 0 for $j \notin A(i)$. The difference is that addition and minimization in the Bellman–Ford iteration term (1.16) are replaced by multiplication and summation, respectively, in the matrix–vector product term (1.17). Thus the computational requirements for a single Bellman–Ford iteration are essentially identical with those for the matrix–vector product Ax. The algorithms and results of Subsection 1.3.6 apply except for the fact that in that subsection, we assumed that A is a fully dense matrix, whereas here A has a sparsity structure determined by the neighbor node sets $A(i)$.

Assume first that we have a system of p processors, with p less than or equal to the number of nodes n, and for simplicity assume that n is divisible by p. It is then natural to let the jth processor update the distance estimates of nodes $(j-1)k+1$ through jk, where $k = n/p$. Let us assume that the jth processor holds the vector x and the lengths a_{ij} for all $j \in A(i)$, and $i = (j-1)k+1$ through $i = jk$. Then the updating of the corresponding coordinates of x according to the Bellman–Ford iteration takes time $O(kr) = O(nr/p)$ for each processor, where $r = \max_{i \neq 1} |A(i)|$ [cf. Eq. (1.15)]. To communicate the results of the updating to the other processors, a multinode broadcast of packets, each containing k numbers, is necessary. If a linear array is used for communication, the results of Subsection 1.3.4 show that the multinode broadcast takes $O(kp) = O(n)$ time. The total time per iteration is then $O\big(\max(n, nr/p)\big)$. If instead a hypercube with p processors is used, the multinode broadcast time becomes $O(n/\log p)$, and the total time per iteration becomes $O\big(\max(n/\log p, nr/p)\big)$. Note that as the maximum number of outgoing arcs r becomes larger, the ratio of computation to communication time increases, and the communication penalty becomes less significant. This means that as r becomes larger, more processors can be fruitfully employed to solve the problem without incurring prohibitive delays due to communications, thereby resulting in higher speedup.

Assume now that a hypercube with n^2 processors arranged in an $n \times n$ array is available, where n is a power of 2. Assume also that at the beginning of each iteration,

each processor (i, j), with $j \in A(i)$, holds x_j and a_{ij}. Then, based on the relationship of the Bellman–Ford iteration with the matrix–vector multiplication described previously, the $O(\log n)$ algorithm for matrix–vector multiplication of Fig. 1.3.26 in Subsection 1.3.6 applies.

We have considered so far a single destination. In the all–pairs version of the problem, we want to find a shortest path from every node to every other node. For this purpose we can apply the Bellman–Ford algorithm separately for each destination. Suppose that p processors ($p \leq n$) are available and assume for simplicity that n is divisible by p. Then we can assign n/p destinations to each processor and apply the (serial) Bellman–Ford algorithm for each of these destinations. This requires $O(|A|n/p)$ time per iteration. (We ignore here the possibility that the shortest distances corresponding to one destination may provide useful information about the shortest distances corresponding to other destinations [FNP81].)

Suppose next that a mesh of n^2 processors is available. The mesh can emulate n independent linear arrays, each having n processors. We can use each of these arrays to solve in parallel a different single destination shortest path problem, in time $O(n)$ per iteration (based on the linear array result given earlier for $p = n$). It is similarly seen that based on the single destination results for a hypercube with $p = n$ and $p = n^2$ given earlier, we can solve the all–pairs problem in $O\big(\max(n/\log n, r)\big)$ time per iteration using a hypercube with $p = n^2$ processors, and in $O(\log n)$ time per iteration using a hypercube with $p = n^3$ processors.

To obtain upper bounds on the running time of the Bellman–Ford algorithm, we should multiply the times per iteration given above with the number of required iterations for termination. This number is $m^* + 1$ for the case of the initial conditions $x_i = \infty$, $i \neq 1$, where m^* is the maximum number of arcs in a shortest path [cf. Prop. 1.1(c)]. In practice, m^* is often much smaller than its upper bound $n - 1$, so one cannot accurately predict the running time for the algorithm without additional knowledge about the problem at hand. In the next subsection, we describe several other algorithms for the all–pairs problem, and we compare them with the Bellman–Ford method.

4.1.2 Other Parallel Shortest Path Methods

Consider the all–pairs shortest path problem where we want to find a shortest path from each node to each other node. We first consider the possibility of assigning a processor to each destination and applying a good serial algorithm to the corresponding single destination problem. Thus, if T_S is the serial time complexity of the single destination algorithm, we obtain a T_S time complexity with n processors, assuming the required problem data are available at each processor. When all arc lengths are nonnegative, we can use Dijkstra's method; this is a popular method for solving the single destination shortest path problem with nonnegative arc lengths (see, e.g., [PaS82]). There are implementations of this method that take time $O(|A| + n \log n)$ (see [FrT84]). For problems where some of the arc lengths are negative, a preprocessing phase is required to transform the problem into a shortest path problem with nonnegative arc lengths. This can be done by replacing a_{ij} with

$$a'_{ij} = a_{ij} + p_j - p_i$$

where p_i is a set of numbers such that $a'_{ij} \geq 0$ (see Exercise 1.3). Finding such a set of numbers is equivalent to solving an assignment problem (see Exercise 1.3 in Section 5.1), and can be done in $O\left(n^{1/2}|A|\log(nC)\right)$ time, where $C = \max_{(i,j)} |a_{ij}|$ and the lengths a_{ij} are assumed integer (see [GaT87] and Exercise 4.5 in Section 5.4). Note that to execute the algorithm, a processor must know the lengths of all arcs, whereas in the Bellman–Ford algorithm, it requires only the lengths of its incident arcs. To broadcast all arc lengths from some node to all other nodes over an optimally chosen spanning tree takes $O(d + |A|) = O(|A|)$ time (by sending each arc length in a separate packet and pipelining the packets, cf. Exercise 3.19 in Section 1.3), where d is the diameter of the interconnection network. Hence, the communication time is of order that is comparable to the order of the computation time. Thus, the time to solve the all–pairs shortest path problem using Dijkstra's method and a network of n processors, each handling a different destination, is $O(|A|+n\log n)$ if all arc lengths are nonnegative, and $O\left(n^{1/2}|A|\log(nC)\right)$ otherwise. The timing estimate obtained earlier for the Bellman–Ford algorithm using n processors is $O\big(m^*|A|\big)$, so it is seen that the preceding estimate for Dijkstra's method is superior when all arc lengths are nonnegative; the situation is less clear when some arc lengths are negative.

We now discuss two algorithms that are specially designed for the all–pairs problem, and have a worst case serial running time which is better than the one of the Bellman–Ford method. These algorithms are not any faster when applied to the single destination problem, and are better suited for dense rather than sparse graphs, as they cannot take advantage of sparsity. By contrast, the serial version of the Bellman–Ford algorithm is speeded up by a factor of n when applied to the single destination problem, and it is also speeded up when the graph is sparse. Furthermore, the Bellman–Ford algorithm has an additional advantage in that it is naturally suited to distributed systems where the processor interconnection network coincides with the graph of the problem, and it also admits an asynchronous implementation (see Section 6.4).

In the following complexity analysis, we consider a message–passing system and we assume that the transmission times of all packets along any link of the processor interconnection network are the same, and are also nonnegligible. Furthermore, we assume that each processor can compute, and simultaneously transmit and receive on all its incident links. The Connectivity and Positive Cycle Assumptions 1.1 and 1.2 are in effect for each destination, but we relax the assumption that a destination node has no outgoing arcs.

The *Floyd–Warshall* algorithm starts with the initial condition

$$x^0_{ij} = \begin{cases} a_{ij}, & \text{if } j \in A(i), \\ \infty, & \text{otherwise,} \end{cases}$$

and generates sequentially for all $k = 0, 1, \ldots, n-1$, and all nodes i and j

$$x_{ij}^{k+1} = \begin{cases} \min\Big\{x_{ij}^k,\ x_{i(k+1)}^k + x_{(k+1)j}^k\Big\}, & \text{if } j \neq i, \\ \infty, & \text{otherwise.} \end{cases}$$

An induction argument shows that x_{ij}^k gives the shortest distance from node i to node j using only nodes from 1 to k as intermediate nodes. The serial solution time is $O(n^3)$, which is superior to the corresponding estimate $O(nm^*|A|)$ for the Bellman–Ford algorithm when the graph is dense.

A parallel implementation of a single iteration of the Floyd–Warshall algorithm on a hypercube with n^2 processors can be shown to take $O(\log n)$ time (see Exercise 1.6). Therefore, if the iterations are synchronized using the global synchronization method (i.e., no processor starts a new iteration before all processors complete the previous iteration and the results of the iteration are communicated to the relevant processors, cf. Subsection 1.4.1), the total time for the n iterations is $O(n \log n)$. This is slower by a factor of $\log n$ over the bound $O(n)$ for the time taken by the algorithm when all communication is instantaneous. On the other hand, it is possible to implement the Floyd–Warshall algorithm on a square mesh of n^2 processors (and *a fortiori* on a hypercube of n^2 processors) using the local synchronization method, whereby each processor executes an iteration once it receives all the necessary information (cf. Subsection 1.4.1). This implementation is described in Exercise 1.7 and takes $O(n)$ time to solve the all-pairs problem, which is the same order of time as when communication is instantaneous.

The *doubling algorithm* is given by

$$x_{ij}^1 = \begin{cases} a_{ij}, & \text{if } j \in A(i), \\ 0, & \text{if } i = j, \\ \infty, & \text{otherwise,} \end{cases}$$

$$x_{ij}^{k+1} = \begin{cases} \min_m\{x_{im}^k + x_{mj}^k\}, & \text{if } i \neq j,\ k = 1,2,\ldots,\lceil \log(n-1) \rceil, \\ 0, & \text{if } i = j,\ k = 1,2,\ldots,\lceil \log(n-1) \rceil. \end{cases}$$

An induction argument shows that for $i \neq j$, x_{ij}^k gives the shortest distance from i to j using paths with 2^{k-1} arcs or less. A serial implementation takes $O\big(n^3 \log m^*\big)$ time, where m^* is the maximum number of arcs in a shortest path over all shortest paths [cf. Eq. (1.8)]. This is inferior to the serial running time of the Floyd–Warshall algorithm, and roughly comparable to that of the Bellman–Ford algorithm when the graph is dense and m^* is small.

Regarding a synchronous parallel implementation of the doubling algorithm, we observe that each iteration has the same structure as the multiplication of two $n \times n$ matrices (the multiplications and additions of matrix multiplication are replaced by additions and minimizations, respectively, in the doubling algorithm iteration). Therefore, the types of algorithms for matrix–matrix multiplication given in Subsection 1.3.6 apply. In particular, we see that the doubling algorithm iteration takes time $O(n)$ using an $n \times n$ mesh of processors and an algorithm that is very similar to the one of Fig. 1.3.27 in Subsection 1.3.6. The total time for the algorithm is $O(n \log m^*)$. It is also possible to implement the doubling algorithm in $O\big((\log n)(\log m^*)\big)$ time using a hypercube of n^3

processors, and the ideas of the matrix–matrix multiplication algorithm of Fig. 1.3.28 in Subsection 1.3.6. This is the minimum known order of solution time for the all–pairs problem, even assuming that communication is instantaneous (a smaller order of solution time can be obtained with a different model of computation than the one we have been using [Kuc82]).

Table 1.1 provides a comparison of the Bellman–Ford, Floyd–Warshall, and doubling algorithms. The estimates given are based on specific implementations of each method, and have either been derived earlier or can be inferred from those derived earlier. For example, the $O(n^3/p)$ estimate for the Floyd–Warshall using a linear array of p processors follows from the $O(n)$ estimate for an $n \times n$ mesh, since each processor of the linear array can emulate n^2/p mesh processors with a slowdown factor of at most n^2/p. The entries of the table suggest that the Bellman–Ford algorithm is superior for the single destination problem, particularly when m^* is relatively small; the Floyd–Warshall and the doubling algorithms are designed for the all–pairs problem, and they are not any faster when applied to the single destination problem. For the all–pairs problem, the Floyd–Warshall is superior to the doubling algorithm when a linear array or an $n \times n$ mesh is used, but the doubling algorithm achieves the smallest known order of solution time when a hypercube with n^3 processors is used. Under some circumstances, where m^* is small and the problem is sparse, the Bellman–Ford algorithm is superior to the Floyd–Warshall using a linear array with $p \leq n$ processors.

Table 1.1: Solution times of shortest path algorithms using various interconnection networks. The times for the single destination problem are the same as for the all–pairs problem for both the Floyd–Warshall and the doubling algorithms. Here n is the number of nodes, r is the maximum number of outgoing arcs from a node [cf. Eq. (1.15)], p is the number of processors, and m^* is the maximum number of arcs in a shortest path [cf. Eq. (1.8)].

Problem	Network	Bellman–Ford	Floyd–Warshall	Doubling
Single Destination	Linear Array $p \leq n$	$O\left(m^* \max\left(n, \frac{nr}{p}\right)\right)$		
	Hypercube $p \leq n$	$O\left(m^* \max\left(\frac{n}{\log p}, \frac{nr}{p}\right)\right)$		
	Hypercube $p = n^2$	$O(m^* \log n)$		
All–Pairs	Linear Array $p \leq n$	$O\left(\frac{m^* \lvert A \rvert n}{p}\right)$	$O\left(\frac{n^3}{p}\right)$	$O\left(\frac{n^3 \log m^*}{p}\right)$
	Mesh $p = n^2$	$O(m^* n)$	$O(n)$	$O(n \log m^*)$
	Hypercube $p = n^2$	$O\left(m^* \max\left(\frac{n}{\log n}, r\right)\right)$		
	Hypercube $p = n^3$	$O(m^* \log n)$		$O\big((\log n)(\log m^*)\big)$

EXERCISES

1.1. Consider the Bellman–Ford algorithm for the initial conditions $x_i^0 = \infty$, $i \neq 1$. Use Lemma 1.1 to show that we have $x_i^n < x_i^{n-1}$ for some i if and only if there exists a cycle of negative length.

1.2. (Gauss–Seidel Version of the Bellman–Ford Algorithm.) Consider the shortest path problem under the Connectivity and Positive Cycle Assumptions.

(a) Viewing the Bellman–Ford algorithm as a Jacobi relaxation method, construct its Gauss–Seidel version, and show that it terminates finitely for any initial condition vector x^0 with $x_1^0 = 0$. *Hint:* Let $J : \Re^n \mapsto \Re^n$ and $G : \Re^n \mapsto \Re^n$ be the Jacobi and Gauss–Seidel relaxation mappings, respectively. Let u be the vector with all coordinates equal to 1 except for $u_1 = 0$. Show that for any scalar $\gamma > 0$ we have

$$x^* - \gamma u \leq J(x^* - \gamma u) \leq G(x^* - \gamma u) \leq x^* \leq G(x^* + \gamma u) \leq J(x^* + \gamma u) \leq x^* + \gamma u.$$

For any initial condition vector x^0 with $x_1^0 = 0$, consider $\gamma > 0$ such that $x^* - \gamma u \leq x^0 \leq x^* + \gamma u$.

(b) For the case of the initial conditions $x_i^0 = \infty$ for all $i \neq 1$, $x_1 = 0$, show that the Gauss–Seidel version converges at least as fast as the Jacobi version.

(c) Assume that the graph is acyclic. Show that there exists a node relaxation order for which the Gauss–Seidel version converges in a single iteration.

1.3. Let the Connectivity and Positive Cycle Assumptions hold, and consider the shortest path problem with arc lengths a_{ij} and shortest distances x_i^*. Let p_i, $i = 1, \ldots, n$, be any scalars. Consider also the shortest path problem with arc lengths $a'_{ij} = a_{ij} + p_j - p_i$.

(a) Show that the length of every cycle with respect to a_{ij} is equal to its length with respect to a'_{ij}, and, therefore, the Positive Cycle Assumption holds for the arc lengths a'_{ij}.

(b) Show that a path is shortest with respect to a_{ij} if and only if it is shortest with respect to a'_{ij}, and that the shortest distances x'_i with respect to a'_{ij} satisfy

$$x_i^* = x'_i + p_i - p_1, \qquad \forall\, i = 1, \ldots, n.$$

1.4. Let $\tilde{x}_i$ be the shortest distances corresponding to a set of arc lengths $\tilde{a}_{ij}$. Consider the shortest path problem for another set of arc lengths a_{ij} and assume that the Connectivity and Positive Cycle Assumptions 1.1 and 1.2 hold.

(a) Assume that $a_{ij} - \tilde{a}_{ij} \geq 0$ for all arcs (i, j), and that $a_{ij} - \tilde{a}_{ij} > 0$ only for arcs (i, j) such that $\tilde{x}_i < \tilde{a}_{ij} + \tilde{x}_j$. Show that the Bellman–Ford algorithm starting from the initial conditions $x_i^0 = \tilde{x}_i$ for all i terminates in one iteration.

(b) Define for $k = 1, 2, \ldots$

$$N_1 = \{i \mid \text{for some arc } (i, j) \text{ with } a_{ij} > \tilde{a}_{ij} \text{ we have } \tilde{x}_i = \tilde{a}_{ij} + \tilde{x}_j\},$$

$$N_{k+1} = \{i \mid \text{for some arc } (i, j) \text{ with } j \in N_k \text{ we have } \tilde{x}_i = \tilde{a}_{ij} + \tilde{x}_j\}.$$

Show that the Bellman–Ford algorithm starting from the initial conditions $x_i^0 = \tilde{x}_i$ for $i \notin \cup_k N_k$ and $x_i^0 = \infty$ for $i \in \cup_k N_k$ terminates in no more than $m^* + 1$ iterations. *Hint:* Show that for $i \notin \cup_k N_k$, $\tilde{x}_i$ is not smaller than the shortest distance of i with respect to lengths a_{ij}.

1.5. Consider the shortest path problem for the case where the graph is a 2–dimensional grid with mk nodes in each dimension (all links are bidirectional). Consider the Bellman–Ford algorithm on a $m \times m$ mesh array of processors with each processor handling the computations for a $k \times k$ "square" block of nodes. Verify that the parallel computation time

per iteration is $O(k^2)$, and that the corresponding communication time is equal to the time needed to transmit a message carrying k numbers plus overhead over a mesh link. Discuss the implication of this result for the upper bound on speedup imposed by the communication penalty, and quantify the role of packet overhead (cf. the analysis of Subsections 1.3.4 and 1.3.5).

1.6. Consider the following implementation of the Floyd–Warshall algorithm on a hypercube of n^2 processors. We asssume that n is a power of 2 and that the processors are arranged in an $n \times n$ array. At the beginning of the algorithm, each processor (i, j) holds a_{ij}. At the kth iteration, processor (i, j) (with $i \neq j$) computes x_{ij}^k. Show that the computation and the communication times per iteration are $O(1)$ and $O(\log n)$, respectively. Assuming that the computations of an iteration can only start after the previous iteration has been completed, establish that the overall time taken by the algorithm is $O(n \log n)$.

1.7. (Parallel $O(n)$ Implementation of the Floyd–Warshall Algorithm.) Consider the following parallel implementation of the Floyd–Warshall algorithm on a square mesh of n^2 processors using the local synchronization method (cf. Subsection 1.4.1). At the beginning of the algorithm each processor (i, j) with $i \neq j$ holds a_{ij}, where $a_{ij} = \infty$ if $(i, j) \notin A$. Processor (i, j) calculates for $k = 0, 1, \ldots, n-1$,

$$x_{ij}^{k+1} = \begin{cases} \min\Big\{x_{ij}^k,\ x_{i(k+1)}^k + x_{(k+1)j}^k\Big\}, & \text{if } j \neq i, \\ \infty, & \text{otherwise,} \end{cases}$$

once it has calculated x_{ij}^k and has received $x_{i(k+1)}^k$ and $x_{(k+1)j}^k$ from its neighboring nodes. Processor (i, j) transmits x_{ij}^{j-1} to its neighboring nodes $(i, j-1)$ and $(i, j+1)$ (if they exist) immediately upon calculating x_{ij}^{j-1}; transmits x_{ij}^{i-1} to its neighboring nodes $(i-1, j)$ and $(i+1, j)$ (if they exist) immediately upon calculating x_{ij}^{i-1}; transmits to processor $(i, j-1)$ (if it exists) whatever it receives from processor $(i, j+1)$; transmits to processor $(i, j+1)$ (if it exists) whatever it receives from processor $(i, j-1)$; transmits to processor $(i-1, j)$ (if it exists) whatever it receives from processor $(i+1, j)$; and transmits to processor $(i+1, j)$ (if it exists) whatever it receives from processor $(i-1, j)$. Assuming that an addition followed by a comparison requires unit time and that transmission of a single number on any one link also requires unit time, show that the algorithm terminates after $O(n)$ time.

1.8. (Parallel Computation of Minimum Weight Spanning Trees [MaP88].) We are given an undirected graph $G = (N, A)$ with a set of nodes $N = \{1, 2, \ldots, n\}$, and a scalar a_{ij} for each arc (i, j) $(a_{ij} = a_{ji})$ called the *weight* of (i, j). The *weight of a spanning tree* of G is defined to be the sum of its arc weights. We refer to the maximum over all the weights of the arcs contained in a given walk as the *critical weight* of the walk. Consider the problem of finding a minimum weight spanning tree (MST).

(a) Show that an arc (i, j) belongs to an MST if and only if the critical weight of every walk starting at i and ending at j is greater than or equal to a_{ij}.

(b) Consider the following iterative algorithm:

$$x_{ij}^0 = \begin{cases} a_{ij}, & \text{if } (i, j) \in A, \\ \infty, & \text{otherwise,} \end{cases}$$

$$x_{ij}^{k+1} = \begin{cases} \min\Big\{x_{ij}^k,\ \min\big\{x_{i(k+1)}^k, x_{(k+1)j}^k\big\}\Big\}, & \text{if } j \neq i, \\ \infty, & \text{otherwise.} \end{cases}$$

Show that x_{ij}^k is the minimum over all critical weights of walks that start at i, end at j, and use only nodes 1 to k as intermediate nodes.

(c) Show how to compute x_{ij}^n in $O(n)$ time using a square mesh with n^2 processors, assuming that processor (i, j) holds initially a_{ij}. *Hint:* Observe the similarity with the Floyd–Warshall algorithm and use Exercise 1.7.

1.9. (Transitive Closure of a Boolean Matrix.) Consider a Boolean $n \times n$ matrix B, that is, a matrix with elements b_{ij} being either zero or one. The transitive closure of B is the matrix B^* obtained by repeated multiplication of $I+B$ with itself, that is, $B^* = \lim_{k\to\infty}(I+B)^k$, where I is the identity matrix, and binary addition and multiplication are replaced by logical OR and logical AND, respectively. Consider the directed graph G with nodes $1, 2, \ldots, n$, and the arc set $\{(i, j) \mid i \neq j \text{ and } b_{ij} = 1\}$.

(a) Show that the ijth element of $(I+B)^k$ is 1 if and only if either $i = j$ or else $i \neq j$ and there exists a directed path from i to j in G having k arcs or less.

(b) Use the result of (a) to show that $B^* = (I+B)^k$ for all k greater or equal to some $\bar{k}$, and characterize $\bar{k}$ in terms of properties of the graph G.

(c) Show how B^* can be computed with a variation of the Floyd–Warshall algorithm.

1.10. (Shortest Path Calculation by Forward Search.) Consider the problem of finding a shortest path from a single origin s to the destination 1. We assume that the Connectivity Assumption 1.1 holds and furthermore $a_{ij} \geq 0$ for all $(i, j) \in A$. The following algorithm makes use of a set of nodes L and of a scalar d_i for each node i. Initially $L = \{s\}$, $d_i = \infty$ for all $i \neq s$, and $d_s = 0$. Let $h_i \geq 0$ be a known underestimate of the shortest distance from node $i \neq 1$ to the destination 1 and let $h_1 = 0$. (One can always take $h_i = 0$ for all i, but the knowledge of sharper underestimates is beneficial.) The algorithm consists of the following steps:

Step 1: Let M be the set of nodes j with $j \in A(i)$ for at least one $i \in L$, and let $\tilde{M}$ be the set

$$\tilde{M} = \left\{ j \in M \mid \min_{\{i \in L \mid j \in A(i)\}} (d_i + a_{ij}) < \min\{d_j, d_1 - h_j\} \right\}.$$

Set $d_j = \min_{\{i \in L \mid j \in A(i)\}} (d_i + a_{ij})$ for all $j \in \tilde{M}$, and set $L = \{j \in \tilde{M} \mid j \neq 1\}$.

Step 2: If L is empty stop; else go to Step 1.

(a) Show that the algorithm terminates, and that upon termination d_1 is equal to the shortest distance from s to 1.

(b) Show that the conclusion of (a) holds for the (Gauss–Seidel) version of the algorithm where Step 1 is replaced by the following:

Step 1: Remove a node i from L. For each $j \in A(i)$, if $d_i + a_{ij} < \min\{d_j, d_1 - h_j\}$ set $d_j = d_i + a_{ij}$, and if $j \neq 1$ and j is not in L, place j in L.

(c) Supplement the algorithms of (a) and (b) with a procedure that allows the construction of a shortest path following termination.

Hint: Show that for both algorithms of parts (a) and (b), the scalar d_i is at all times either ∞ or the length of some path from s to i. Use this fact to show that the algorithms terminate finitely.

4.2 MARKOV CHAINS WITH TRANSITION COSTS

In this section, we lay the groundwork for the formulation of stochastic dynamic programming models, which will be introduced in the next section.

Consider a stationary discrete–time Markov chain with state space

$$S = \{1, 2, \ldots, n\},$$

and transition probability matrix P with elements p_{ij} (see Appendix D for a review of Markov chains). Suppose that if the state is $s(t) = j$ at time t, there is a cost $\alpha^t c_j$ incurred, where c_j and α are given scalars. We assume that $0 < \alpha \leq 1$, and we refer to α as the *discount factor*. The implication here is that the cost for being at state j at some time is reduced by a factor α over the cost for being at j one period earlier. Thus, when $\alpha < 1$, costs accumulate primarily during the early periods. Since we have

$$\Pr\big(s(t) = j \mid s(0) = i\big) = [P^t]_{ij},$$

the total expected cost associated with the system starting at state i is

$$x_i^* = \sum_{t=0}^{\infty} \alpha^t E\left[c_{s(t)} \mid s(0) = i\right] = \sum_{t=0}^{\infty} \alpha^t \left[P^t c\right]_i .$$

Equivalently, we have

$$x^* = \sum_{t=0}^{\infty} \alpha^t P^t c, \tag{2.1}$$

where x^* and c are the vectors with coordinates x_i^* and c_i, respectively. In this section, we show, under reasonable assumptions, that the series defining x^* in Eq. (2.1) is convergent and that $x^* = c + \alpha P x^*$. We then consider the iteration

$$x := c + \alpha P x,$$

which can be implemented in both Gauss–Seidel and Jacobi modes (cf. Chapter 2), and we show that it converges to x^*. We differentiate between the two cases where $0 < \alpha < 1$ and $\alpha = 1$.

Assumption 2.1. $0 < \alpha < 1$.

Under Assumption 2.1 the cost is said to be *discounted*. In this case, the series defining the cost vector x^* in Eq. (2.1) can be shown to be convergent. Indeed, we have

$$\|P\|_\infty = \max_i \sum_{j=1}^{n} p_{ij} = 1,$$

where $\|\cdot\|_\infty$ denotes the maximum norm. Consider the equation $x = c + \alpha Px$. It has a unique solution since $\rho(\alpha P) \leq \alpha\|P\|_\infty = \alpha < 1$. This solution, call it $\tilde{x}$, satisfies

$$\tilde{x} = c + \alpha P(c + \alpha P\tilde{x}) = \cdots = \sum_{t=0}^{m-1} \alpha^t P^t c + \alpha^m P^m \tilde{x}, \qquad \forall m. \tag{2.2}$$

By taking the limit in this equation as $m \to \infty$, we obtain that the series in Eq. (2.1) is convergent and that $\tilde{x} = x^*$. Therefore, the optimal cost vector x^* is well defined and satisfies

$$x^* = c + \alpha Px^*. \tag{2.3}$$

Consider now the iteration

$$x := c + \alpha Px \tag{2.4}$$

for an arbitrary initial condition. Since $\|\alpha P\|_\infty = \alpha$, the function $c+\alpha Px$ is a contraction of modulus α with respect to the maximum norm when $0 < \alpha < 1$. It follows that the iteration $x := c + \alpha Px$ converges to the unique fixed point x^* [cf. Eq. (2.3)] when implemented in a synchronous parallel setting, as discussed in Chapter 2. This is true for both a Jacobi and a Gauss–Seidel mode of implementation (Props. 6.7 and 6.8 in Section 2.6).

We now turn to the case $\alpha = 1$, and consider the question whether the series in Eq. (2.1) defining the cost vector x^* is convergent and whether $x^* = c + Px^*$ [cf. Eq. (2.3)]. The equation $x = c + Px$ may or may not have a solution. If it has a solution x, it will have an infinite number of solutions (for every scalar r, the vector with coordinates $x_i + r$ is also a solution). To delineate situations where a solution exists, we draw motivation from the special case where $c \geq 0$ and we reason as follows. If $c \geq 0$ and for some state i, the cost x_i^* is finite, then starting from i, the system must enter eventually (with probability one) a set of zero–cost states, and stay within that set permanently; otherwise the number of times a positive cost state is visited would be infinite with positive probability, contradicting the finiteness of x_i^*. Therefore, if x_i^* is finite for some i, there is a nonempty subset of the set of zero–cost states,

$$\bar{S} \subset \{j \in S \mid c_j = 0\},$$

which is *absorbing* in the sense

$$p_{jm} = 0, \qquad \forall\, j \in \bar{S},\ m \notin \bar{S}, \tag{2.5}$$

and is such that there is a sequence of positive probability transitions leading from i to at least one state in $\bar{S}$. To guarantee that x_i^* is finite for all i, we consequently lump all states in $\bar{S}$ into a single state, say state 1, and consider the following assumption.

Assumption 2.2. $\alpha = 1$, state 1 is absorbing and cost-free ($p_{11} = 1, c_1 = 0$), and there exists some $\bar{t} > 0$ such that state 1 is reached with positive probability after at most $\bar{t}$ transitions regardless of the initial state ($[P^{\bar{t}}]_{i1} > 0$ for all i).

To establish the equation $x^* = c + Px^*$ under the preceding assumption, we write P in the form

$$P = \begin{bmatrix} 1 & 0 \ldots 0 \\ p_{21} & \\ \vdots & \tilde{P} \\ p_{n1} & \end{bmatrix}. \tag{2.6}$$

It was shown in Prop. 8.4 of Section 2.8 that for some $\delta > 0$, we have

$$\rho(\tilde{P}^{\bar{t}}) \leq \|\tilde{P}^{\bar{t}}\|_\infty \leq 1 - \delta. \tag{2.7}$$

Since $c_1 = 0$, it is seen from Eq. (2.6) that for all $t \geq 0$, we have

$$P^t c = \begin{bmatrix} 0 \\ \tilde{P}^t \tilde{c} \end{bmatrix}, \tag{2.8}$$

where

$$\tilde{c} = \begin{bmatrix} c_2 \\ \vdots \\ c_n \end{bmatrix}.$$

From Eq. (2.7) we see that the spectral radius $\rho(\tilde{P})$ of $\tilde{P}$ satisfies $\rho(\tilde{P}) < 1$. Therefore the equation $\tilde{x} = \tilde{c} + \tilde{P}\tilde{x}$ has a unique solution. This solution, call it $\tilde{x}^*$, satisfies

$$\tilde{x}^* = \tilde{c} + \tilde{P}(\tilde{c} + \tilde{P}\tilde{x}^*) = \cdots = \sum_{t=0}^{m-1} \tilde{P}^t \tilde{c} + \tilde{P}^m \tilde{x}^*, \qquad \forall m. \tag{2.9}$$

Since $\rho(\tilde{P}) < 1$, we have $\lim_{m\to\infty} \tilde{P}^m \tilde{x}^* = 0$. Therefore, by taking the limit in Eq. (2.9) as $m \to \infty$, we obtain that the series $\sum_{t=0}^{\infty} P^t c$ defining the optimal cost vector x^* is convergent and that

$$\tilde{x}^* = \begin{bmatrix} x_2^* \\ \vdots \\ x_n^* \end{bmatrix}.$$

Since $\tilde{x}^* = \tilde{c} + \tilde{P}\tilde{x}^*$, $c_1 = 0$, and $x_1^* = 0$, it follows using Eq. (2.6) that x^* satisfies the equation $x^* = c + Px^*$. Furthermore, since $\rho(\tilde{P}) < 1$, the iteration

$$\tilde{x} := \tilde{c} + \tilde{P}\tilde{x},$$

converges to $\tilde{x}^*$ starting from an arbitrary initial condition. In Section 6.3, we will see that this iteration converges even if executed in a totally asynchronous environment.

EXERCISES

2.1. **(Upper and Lower Bounds on the Cost Vector.)** Let $0 < \alpha < 1$, let x be an arbitrary vector in $\Re^n$, and let

$$\bar{x} = c + \alpha P x,$$

$$\bar{\gamma} = \max_i (\bar{x}_i - x_i), \qquad \gamma = \min_i (\bar{x}_i - x_i).$$

Show that

$$x + \frac{\gamma}{1-\alpha} e \leq \bar{x} + \frac{\alpha\gamma}{1-\alpha} e \leq x^* \leq \bar{x} + \frac{\alpha\bar{\gamma}}{1-\alpha} e \leq x + \frac{\bar{\gamma}}{1-\alpha} e,$$

where x^* is the unique solution of the system $x = c + \alpha P x$, and e is the vector $(1, 1, \ldots, 1)$.

2.2. Let Assumption 2.2 hold, and let t_i denote the mean first passage time from state i to state 1, that is, the average number of steps required to reach state 1 starting from state i. Show that these times are the unique solution of the system of equations

$$t_i = 1 + \sum_{j=2}^{n} p_{ij} t_j, \qquad i = 2, \ldots, n,$$

$$t_1 = 0.$$

4.3 MARKOVIAN DECISION PROBLEMS

We now generalize the problem of the previous section by allowing the transition probability matrix at each stage to be subject to optimal choice. At each state i, we are given a finite set of decisions or controls $U(i)$. If the state is i and control u is chosen at time t, the cost incurred is $\alpha^t c_i(u)$, where $\alpha > 0$, $c_i(u)$ are given scalars; the system then moves to state j with given probability $p_{ij}(u)$. Consider the finite set of functions μ that map states i into controls $\mu(i) \in U(i)$, that is, the set

$$M = \{\mu \mid \mu(i) \in U(i),\ i = 1, \ldots, n\}.$$

We can identify each μ with a rule for choosing a control as a function of the state. Similarly we identify a sequence $\{\mu_0, \mu_1, \dots\}$ ($\mu_t \in M$ for all t), with a rule for choosing at time t and at state i the control $\mu_t(i)$. Such a sequence is called a *policy* and if μ_t is the same for all t, it is called a *stationary policy*.

Let $P(\mu)$ be the transition probability matrix corresponding to μ, that is, the matrix with elements

$$[P(\mu)]_{ij} = p_{ij}\big(\mu(i)\big), \qquad i, j = 1, \dots, n.$$

Define also the cost vector $c(\mu)$ corresponding to $\mu \in M$,

$$c(\mu) = \begin{bmatrix} c_1\big(\mu(1)\big) \\ \vdots \\ c_n\big(\mu(n)\big) \end{bmatrix}.$$

For any policy $\pi = \{\mu_0, \mu_1, \dots\}$, we have

$$\Pr\big(\text{State is } j \text{ at time } t \mid \text{Initial state is } i, \text{ and } \pi \text{ is used}\big) = \big[P(\mu_0)P(\mu_1)\dots P(\mu_{t-1})\big]_{ij}.$$

Therefore, if $x_i(\pi)$ is the expected cost corresponding to initial state i and policy $\pi = \{\mu_0, \mu_1, \dots\}$, and $x(\pi)$ is the vector with coordinates $x_1(\pi), \dots, x_n(\pi)$, we have

$$x(\pi) = \sum_{t=0}^{\infty} \alpha^t \big[P(\mu_0)P(\mu_1)\cdots P(\mu_{t-1})\big] c(\mu_t), \tag{3.1}$$

assuming the above series converges. Conditions for convergence will be introduced shortly. When the above series is not known to converge, we use the definition

$$x(\pi) = \limsup_{k\to\infty} \sum_{t=0}^{k} \alpha^t \big[P(\mu_0)P(\mu_1)\cdots P(\mu_{t-1})\big] c(\mu_t). \tag{3.2}$$

For a stationary policy $\{\mu, \mu, \dots\}$, the corresponding cost vector is denoted

$$x(\mu) = \limsup_{k\to\infty} \sum_{t=0}^{k} \alpha^t P(\mu)^t c(\mu). \tag{3.3}$$

We define the optimal expected cost starting at state i as

$$x_i^* = \inf_{\pi} x_i(\pi), \qquad \forall\, i. \tag{3.4}$$

We say that the policy π^* is optimal if

$$x_i(\pi^*) = \inf_{\pi} x_i(\pi), \qquad \forall\, i.$$

It is convenient to introduce the mappings $T_\mu : \Re^n \mapsto \Re^n$ and $T : \Re^n \mapsto \Re^n$ defined by

$$T_\mu(x) = c(\mu) + \alpha P(\mu)x, \qquad \mu \in M, \tag{3.5}$$

$$T(x) = \min_{\mu \in M} \left[c(\mu) + \alpha P(\mu)x\right] = \min_{\mu \in M} T_\mu(x). \tag{3.6}$$

The minimization in the preceding equation is meant to be separate for each coordinate, that is, the ith coordinate of $T(x)$ is

$$\left[T(x)\right]_i = \min_{u \in U(i)} \left[c_i(u) + \alpha \sum_{j=1}^{n} p_{ij}(u)x_j\right]. \tag{3.7}$$

Note that T_μ is the mapping involved in the iteration $x := c + \alpha Px$ of the previous section, with c and P replaced by $c(\mu)$ and $P(\mu)$, respectively. A straightforward calculation verifies that for all $k \geq 1$ and x

$$(T_{\mu_0}T_{\mu_1}\cdots T_{\mu_k})(x) = \alpha^{k+1}P(\mu_0)P(\mu_1)\cdots P(\mu_k)x + \sum_{t=0}^{k} \alpha^t \left[P(\mu_0)P(\mu_1)\cdots P(\mu_{t-1})\right]c(\mu_t)$$

where $(T_{\mu_0}T_{\mu_1}\cdots T_{\mu_k})$ is the composition of the mappings $T_{\mu_0}, \ldots, T_{\mu_k}$. Therefore, we can write [cf. Eqs. (3.1)–(3.3)]

$$x(\pi) = \limsup_{k \to \infty} (T_{\mu_0}T_{\mu_1}\cdots T_{\mu_k})(x^0), \tag{3.8}$$

$$x(\mu) = \limsup_{k \to \infty} T_\mu^k(x^0), \tag{3.9}$$

where x^0 is the zero vector

$$x^0 = (0, 0, \ldots, 0),$$

and T_μ^k is the composition of T_μ with itself k times.

Our main result will be to show, under reasonable conditions, that the optimal cost vector x^* is a fixed point of the mapping T, that is,

$$x^* = T(x^*),$$

and that x^* can be obtained in the limit through the iteration

$$x := T(x),$$

known as the *dynamic programming* iteration.

The following proposition gives some basic properties of T and T_μ.

Proposition 3.1. The following hold for the mappings T and T_μ of Eqs. (3.5) and (3.6):

(a) T and T_μ are monotone in the sense

$$x \le x' \quad \Rightarrow \quad T(x) \le T(x'),$$
$$x \le x' \quad \Rightarrow \quad T_\mu(x) \le T_\mu(x'), \qquad \forall \mu \in M.$$

(b) For all $x \in \Re^n$, scalars r, integers $t > 0$, and functions $\mu_1, \ldots, \mu_t \in M$, we have

$$T^t(x + re) = T^t(x) + \alpha^t re,$$
$$(T_{\mu_1} T_{\mu_2} \cdots T_{\mu_t})(x + re) = (T_{\mu_1} T_{\mu_2} \cdots T_{\mu_t})(x) + \alpha^t re,$$

where e is the vector $(1, 1, \ldots, 1)$.

(c) For all x and $x' \in \Re^n$, we have

$$\|T(x) - T(x')\|_\infty \le \alpha \|x - x'\|_\infty,$$
$$\big\|T_\mu(x) - T_\mu(x')\big\|_\infty \le \alpha \|x - x'\|_\infty, \qquad \forall \mu \in M.$$

In particular, if $\alpha < 1$, then T and T_μ are contraction mappings with respect to the maximum norm $\|\cdot\|_\infty$.

Proof. Assertion (a) is obtained using the nonnegativity of the elements of $\alpha P(\mu)$. Assertion (b) is obtained for $t = 1$ using the fact $P(\mu)e = e$, and is generalized to arbitrary t using induction. To prove (c), we note that

$$\begin{aligned} T(x) &= \min_\mu \big[c(\mu) + \alpha P(\mu) x' + \alpha P(\mu)(x - x')\big] \\ &= \min_\mu \big[T_\mu(x') + \alpha P(\mu)(x - x')\big] \le T(x') + \alpha \|x - x'\|_\infty e. \end{aligned}$$

Similarly,

$$T(x') \le T(x) + \alpha \|x - x'\|_\infty e.$$

Combining these two inequalities, we obtain

$$-\alpha \|x - x'\|_\infty e \le T(x) - T(x') \le \alpha \|x - x'\|_\infty e,$$

which shows that

$$\|T(x) - T(x')\|_\infty \le \alpha \|x - x'\|_\infty.$$

A similar argument shows that $\left\|T_\mu(x) - T_\mu(x')\right\|_\infty \leq \alpha\|x - x'\|_\infty$ for all $\mu \in M$. **Q.E.D.**

4.3.1 Discounted Problems

Consider now the case where there is a discount factor.

Assumption 3.1. (*Discounted Cost*) $0 < \alpha < 1$.

The main results follow from the contraction property of Prop. 3.1(c).

Proposition 3.2. Under the Discounted Cost Assumption 3.1 the following hold:

(a) The optimal cost vector x^* is the unique fixed point of T within $\Re^n$.

(b) For every $x \in \Re^n$ and $\mu \in M$, there holds

$$\lim_{t\to\infty} T^t(x) = x^*, \qquad \lim_{t\to\infty} T^t_\mu(x) = x(\mu),$$

and the convergence is geometric.

(c) A stationary policy $\{\mu^*, \mu^*, \ldots\}$ is optimal if and only if

$$T_{\mu^*}(x^*) = T(x^*).$$

Proof.

(a) By Prop. 3.1(c), T is a contraction mapping and hence has a unique fixed point $\tilde{x}$. We will show that $\tilde{x} = x^*$. Let x^0 be the zero vector. For any policy $\pi = \{\mu_0, \mu_1, \ldots\}$, we see from the definition (3.6) of T that $T(x^0) \leq T_{\mu_t}(x^0)$, and by using the monotonicity of T, we obtain

$$T^2(x^0) \leq (TT_{\mu_t})(x^0) \leq (T_{\mu_{t-1}}T_{\mu_t})(x^0).$$

Proceeding similarly, we obtain for all t,

$$T^{t+1}(x^0) \leq (T_{\mu_0}T_{\mu_1}\cdots T_{\mu_t})(x^0).$$

By taking the limit superior as $t \to \infty$, and using the definition (3.8) of $x(\pi)$, and the fact $\lim_{t\to\infty} T^{t+1}(x^0) = \tilde{x}$, we obtain for all π

$$\tilde{x} \leq x(\pi).$$

Therefore,

$$\tilde{x} \leq x^*. \tag{3.10}$$

To prove the reverse inequality, we select a function $\mu \in M$ such that

$$T_\mu(\tilde{x}) = T(\tilde{x}).$$

Applying T_μ repeatedly and using the fact $T(\tilde{x}) = \tilde{x}$ we obtain

$$T^t_\mu(\tilde{x}) = \tilde{x}.$$

We showed in the previous section that $\lim_{t\to\infty} T^t_\mu(\tilde{x}) = x(\mu)$, so we must have

$$x(\mu) = \tilde{x}.$$

Since $x^* \le x(\mu)$, we obtain $x^* \le \tilde{x}$, which, combined with the inequality $x^* \ge \tilde{x}$ proved earlier, shows that $x^* = \tilde{x}$.

(b) This follows from the contraction property of T and T_μ, and part (a).

(c) Suppose that $T_{\mu^*}(x^*) = T(x^*)$. Using part (a), we have $T(x^*) = x^*$. It follows that $T_{\mu^*}(x^*) = x^*$, and by the analysis of the previous section, we obtain $x(\mu^*) = x^*$, so $\{\mu^*, \mu^*, \ldots\}$ is optimal. Conversely, suppose $\{\mu^*, \mu^*, \ldots\}$ is optimal. Then x^* is the cost corresponding to μ^*, so x^* is the unique fixed point of T_{μ^*} as well as the unique fixed point of T [by part (a)]. Hence, $T_{\mu^*}(x^*) = x^* = T(x^*)$. **Q.E.D.**

Note that Prop. 3.2(b) guarantees the validity of the dynamic programming algorithm that starts from an arbitrary vector x and successively generates $T(x)$, $T^2(x), \ldots$. This algorithm yields in the limit x^*, and from x^* one can obtain an optimal stationary policy using Prop. 3.2(c) (see also Exercise 3.1). The contraction property of T guarantees also that the Gauss–Seidel algorithm based on T converges to x^* (Prop. 1.4 in Section 3.1).

4.3.2 Undiscounted Problems—Stochastic Shortest Paths

When $\alpha = 1$, the problem is of interest primarily when there is a cost–free state, say state 1, which is absorbing (this is similar to the case of a single policy in the previous section). The objective then is to reach this state at minimum expected cost. We say that a stationary policy $\{\mu, \mu, \ldots\}$ is *proper* if $\lim_{t\to\infty} \left[P^t(\mu)\right]_{i1} = 1$ for all $i \in S$; otherwise, we say that π is *improper*. We will operate under the following assumption:

Assumption 3.2. (*Undiscounted Cost*)$\alpha = 1$, $p_{11}(u) = 1$, and $c_1(u) = 0$ for all $u \in U(1)$, and there exists at least one proper stationary policy. Furthermore, each improper stationary policy yields infinite cost for at least one initial state, that is, for each improper $\{\mu, \mu, \ldots\}$, there is a state i such that $\limsup_{k\to\infty} \left[\sum_{t=0}^k P^t(\mu)c(\mu)\right]_i = \infty$.

The shortest path problem of Section 4.1 is an important example of a dynamic programming problem where the above assumption holds. To establish this fact, we view the nodes of the given graph as the states of a Markov chain. A stationary policy

$\{\mu, \mu, \ldots\}$ is identified with a rule that assigns to each node $i \neq 1$ a neighbor node $\mu(i) = j$ with $(i,j) \in A(i)$. Given such a μ, the transitions are deterministic and the cost of the transition at i is $a_{i\mu(i)}$ (see Fig. 4.3.1). The destination node 1 has no outgoing arcs by assumption, and is viewed as an absorbing and cost–free state for all μ. For each initial state $i \neq 1$ and each stationary policy, there are two possibilities. The first is that the sequence of generated states does not contain state 1, in which case the policy is improper; in this case, the Positive Cycle Assumption 1.2, implies that this state is associated with infinite cost. The second possibility is that the policy is proper, so the system eventually reaches state 1 and subsequently stays there, in which case, the total cost equals the sum of the transition costs up to the time state 1 is reached first. In this case, the sequence of transitions defines a simple path starting at i and ending at 1, and the corresponding total cost equals the length of the path. It is seen, therefore, that optimal stationary policies are those that correspond to shortest paths. The Connectivity Assumption 1.1 is seen to be equivalent to the existence of at least one proper policy, while the Positive Cycle Assumption 1.2 is seen to be equivalent to the existence of an infinite cost state for every improper policy. Therefore, the assumptions made in Section 4.1 are, in effect, equivalent with the Undiscounted Cost Assumption 3.2 as applied to the shortest path problem. We may view the general problem under Assumption 3.2 as a stochastic version of the shortest path problem whereby at any given node, instead of choosing a successor node, we choose a probability distribution over the successor nodes with the objective of minimizing the expected length of the path that will be traveled from the given node to the destination node 1.

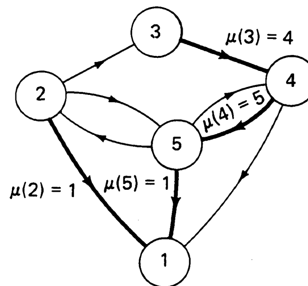

Figure 4.3.1 Viewing a shortest path problem as a (deterministic) dynamic programming problem. The figure shows the paths corresponding to a stationary policy $\{\mu, \mu, \ldots\}$. The control $\mu(i)$ associated with a node $i \neq 1$ is a node adjacent to i. The policy shown is proper because the path from every state leads to the destination.

Exercise 3.2 explores conditions under which the Undiscounted Cost Assumption 3.2 is guaranteed to hold. If *all* stationary policies are proper, it can be shown (Exercise 3.3) that the mapping T is a contraction over the subspace $X = \{x \in \Re^n \mid x_1 = 0\}$ with respect to some weighted maximum norm. This is not true in general, however, as the example of Fig. 4.3.2 shows. Despite this fact, essentially the same results as for the Discounted Cost case hold (cf. Prop. 3.2).

Proposition 3.3. Let the Undiscounted Cost Assumption 3.2 hold. Then:

(a) The optimal cost vector x^* is the unique fixed point of T within the subspace

$$X = \{x \in \Re^n \mid x_1 = 0\}.$$

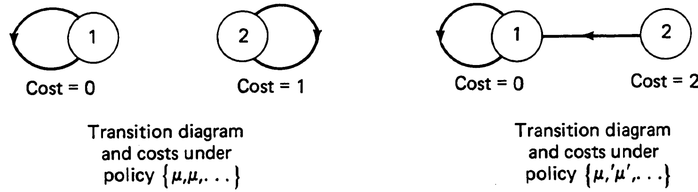

Figure 4.3.2 Example problem where the Undiscounted Cost Assumption 3.2 is satisfied, but the mapping T is not a contraction mapping over the subspace $X = \{x \in \Re^n \mid x_1 = 0\}$. Here $M = \{\mu, \mu'\}$ with transition probabilities and costs as shown. The mapping T over the set $X = \{(x_1, x_2) \mid x_1 = 0\}$ is given by

$$[T(x)]_1 = 0,$$

$$[T(x)]_2 = \min\{1 + x_2, 2\}.$$

Thus for $x = (0, x_2)$ and $x' = (0, x_2')$, with $x_2 < 1$ and $x_2' < 1$ we have

$$\left|[T(x)]_2 - [T(x')]_2\right| = |(1 + x_2) - (1 + x_2')| = |x_2 - x_2'|.$$

Therefore, T is not a contraction mapping with respect to any weighted maximum norm.

(b) For every $x \in X$, there holds

$$\lim_{t \to \infty} T^t(x) = x^*.$$

(c) A stationary policy $\pi = \{\mu^*, \mu^*, \ldots\}$ is optimal if and only if

$$T_{\mu^*}(x^*) = T(x^*).$$

Proof. We first show the following:

Lemma 3.1. Let Assumption 3.2 hold:

(a) If the stationary policy $\{\mu, \mu, \ldots\}$ is proper, then $x(\mu)$ is the unique fixed point of T_μ within X. Furthermore, $\lim_{t \to \infty} T_\mu^t(x) = x(\mu)$ for all $x \in X$.
(b) If $x \geq T_\mu(x)$ for some $x \in X$, then $\{\mu, \mu, \ldots\}$ is proper.

Proof.

(a) If $\{\mu, \mu, \ldots\}$ is proper, then Assumption 2.2 of Section 4.2 is satisfied and the conclusion follows from the results of that section.

(b) If $x \in X$ and $x \geq T_\mu(x)$, then by the monotonicity of T_μ,

$$x \geq T_\mu^t(x) = P^t(\mu)x + \sum_{k=0}^{t-1} P^k(\mu)c(\mu), \qquad \forall\, t \geq 1. \tag{3.11}$$

If $\{\mu, \mu, \ldots\}$ were improper, then some subsequence of $\sum_{k=0}^{t-1} P^k(\mu)c(\mu)$ would have a coordinate that tends to infinity, thereby contradicting the above inequality. **Q.E.D.**

We now return to the proof of Prop. 3.3. We first show that T has at most one fixed point within X. Indeed, if x and x' are two fixed points in X, then we select μ and μ' such that $x = T(x) = T_\mu(x)$ and $x' = T(x') = T_{\mu'}(x')$. By Lemma 3.1(b), we have that $\{\mu, \mu, \ldots\}$ and $\{\mu', \mu', \ldots\}$ are proper, and furthermore $x = x(\mu)$ and $x' = x(\mu')$. We have $x = T^t(x) \leq T_{\mu'}^t(x)$ for all $t \geq 1$, and by Lemma 3.1(a), we obtain $x \leq \lim_{t\to\infty} T_{\mu'}^t(x) = x(\mu') = x'$. Similarly, $x' \leq x$, showing that $x = x'$ and that T has at most one fixed point within X.

We next show that T has a fixed point within X. Let $\{\mu, \mu, \ldots\}$ be a proper policy. Choose $\mu' \in M$ such that $T_{\mu'}\big(x(\mu)\big) = T\big(x(\mu)\big)$. Then we have $x(\mu) = T_\mu\big(x(\mu)\big) \geq T_{\mu'}\big(x(\mu)\big)$. By Lemma 3.1(b), $\{\mu', \mu', \ldots\}$ is proper, and by the monotonicity of $T_{\mu'}$, we obtain

$$x(\mu) \geq \lim_{t\to\infty} T_{\mu'}^t\big(x(\mu)\big) = x(\mu'). \tag{3.12}$$

If $x(\mu) = x(\mu')$, then we obtain $x(\mu) = x(\mu') = T_{\mu'}\big(x(\mu')\big) = T_{\mu'}\big(x(\mu)\big) = T\big(x(\mu)\big)$ and $x(\mu)$ is a fixed point of T. If $x(\mu) \neq x(\mu')$, then, from Eq. (3.12), $x(\mu) \geq x(\mu')$ and $x_i(\mu) > x_i(\mu')$ for at least one state i. We then replace μ by μ' and continue the process. Since the set of proper policies is finite, we must obtain eventually two successive proper policies with equal cost vectors, thereby showing that T has a fixed point within X.

Next we show that the unique fixed point of T within X is equal to the optimal cost vector x^*, and that $T^t(x) \to x^*$ for all $x \in X$. Indeed, the construction of the preceding paragraph provides a proper policy $\{\mu, \mu, \ldots\}$ such that $T_\mu\big(x(\mu)\big) = T\big(x(\mu)\big) = x(\mu)$. We will show that $T^t(x) \to x(\mu)$ for all $x \in X$, and that $x(\mu) = x^*$. Let Δ be the vector with coordinates

$$\Delta_i = \begin{cases} 0, & \text{if } i = 1, \\ \delta, & \text{if } i \neq 1, \end{cases} \tag{3.13}$$

where $\delta > 0$ is some scalar, and let x^Δ be the vector in X satisfying

$$T_\mu(x^\Delta) = x^\Delta - \Delta. \tag{3.14}$$

[There is a unique such vector because the equation $x^\Delta = c(\mu) + \Delta + P(\mu)x^\Delta$ $\big(= T_\mu(x^\Delta) + \Delta\big)$ has a unique solution within X by the analysis of Section 4.2.] Since x^Δ

is the cost vector corresponding to μ for $c(\mu)$ replaced by $c(\mu)+\Delta$, we have $x^{\Delta} \geq x(\mu)$. Furthermore, for any $x \in X$, there exists $\Delta > 0$ such that $x \leq x^{\Delta}$. We have

$$x(\mu) = T\big(x(\mu)\big) \leq T(x^{\Delta}) \leq T_{\mu}(x^{\Delta}) = x^{\Delta} - \Delta \leq x^{\Delta}.$$

Using the monotonicity of T and the previous relation, we obtain

$$x(\mu) = T^t\big(x(\mu)\big) \leq T^t(x^{\Delta}) \leq T^{t-1}(x^{\Delta}) \leq x^{\Delta}, \qquad \forall\, t \geq 1. \tag{3.15}$$

Hence, $T^t(x^{\Delta})$ converges to some $\tilde{x} \in X$, and by continuity of T, we must have $\tilde{x} = T(\tilde{x})$. By the uniqueness of the fixed point of T shown earlier, we must have $\tilde{x} = x(\mu)$. It is also seen using the fact $x_1(\mu) = 0$, that

$$x(\mu) - \Delta = T\big(x(\mu)\big) - \Delta \leq T\big(x(\mu) - \Delta\big) \leq T\big(x(\mu)\big) = x(\mu), \tag{3.16}$$

so $x(\mu) - \Delta \leq \lim_{t\to\infty} T^t\big(x(\mu) - \Delta\big) \leq x(\mu)$. Similarly, as earlier, it follows that $\lim_{t\to\infty} T^t\big(x(\mu) - \Delta\big) = x(\mu)$. For any $x \in X$, we can find $\delta > 0$ such that

$$x(\mu) - \Delta \leq x \leq x^{\Delta}.$$

By monotonicity of T, we then have

$$T^t\big(x(\mu) - \Delta\big) \leq T^t(x) \leq T^t(x^{\Delta}), \qquad \forall\, t \geq 1, \tag{3.17}$$

and since $\lim_{t\to\infty} T^t\big(x(\mu){-}\Delta\big) = \lim_{t\to\infty} T^t(x^{\Delta}) = x(\mu)$, it follows that $\lim_{t\to\infty} T^t(x) = x(\mu)$. To show that $x(\mu) = x^*$, take any policy $\pi = \{\mu_0, \mu_1, \ldots\}$. We have

$$(T_{\mu_0} \cdots T_{\mu_{t-1}})(x^0) \geq T^t(x^0),$$

where x^0 is the zero vector. Taking the limit superior in the preceding inequality, we obtain

$$x(\pi) \geq x(\mu),$$

so $\{\mu, \mu, \ldots\}$ is an optimal policy and $x(\mu) = x^*$.

To prove part (c), we note that if $\{\mu^*, \mu^*, \ldots\}$ is optimal, then $x(\mu^*) = x^*$, so $T_{\mu^*}(x^*) = T_{\mu^*}\big(x(\mu^*)\big) = x(\mu^*) = x^* = T(x^*)$. Conversely, if $x^* = T(x^*) = T_{\mu^*}(x^*)$ it follows from Lemma 3.1 that $\{\mu^*, \mu^*, \ldots\}$ is proper, and by using the results of Section 4.2, we obtain $x^* = x(\mu^*)$. Therefore, $\{\mu^*, \mu^*, \ldots\}$ is optimal. **Q.E.D.**

We note that the convergence property of Prop. 3.3(b) can also be shown for the Gauss–Seidel algorithm based on T (Exercise 3.7). An alternative method of proof for Prop. 3.3 is outlined in Exercise 3.4 under somewhat different assumptions.

The results of Prop. 3.3 may not hold if the Assumption 3.2 or the undiscounted cost model are modified in seemingly minor ways. Figure 4.3.3 gives an example where there is a (nonoptimal) improper policy that yields finite cost for all initial states and Prop. 3.3 fails to hold. Exercise 3.8 gives an example where Prop. 3.3 fails to hold when the set of decisions $U(i)$ is not finite, even though all stationary policies are proper.

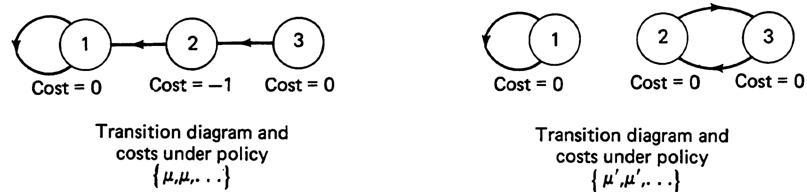

Figure 4.3.3 Example where Prop. 3.3 fails to hold. Here there are two stationary policies, $\{\mu, \mu, \ldots\}$ and $\{\mu', \mu', \ldots\}$ with transition probabilities and costs as shown. The equation $x = T(x)$ over the subspace $X = \{(x_1, x_2, x_3) \mid x_1 = 0\}$ is given by

$$x_1 = 0,$$

$$x_2 = \min\{-1, x_3\},$$

$$x_3 = x_2,$$

and is satisfied by any vector of the form $x = (0, \delta, \delta)$ with $\delta \leq -1$. Here the proper policy $\{\mu, \mu, \ldots\}$ is optimal and the corresponding optimal cost vector is $x^* = (0, -1, -1)$. The difficulty is that there is the nonoptimal improper policy $\{\mu', \mu', \ldots\}$ that has finite (zero) cost for all initial states. This example depends on the existence of a negative state cost (see Exercise 3.4).

The construction used in the proof of Prop. 3.3 to show that T has a fixed point constitutes an algorithm, known as *policy iteration*, for obtaining an optimal proper policy starting with an arbitrary proper policy. In the typical iteration of this algorithm, given a proper policy $\{\mu, \mu, \ldots\}$ and the corresponding cost vector $x(\mu)$, one obtains a new proper policy $\{\mu', \mu', \ldots\}$ satisfying the equation $T_{\mu'}\big(x(\mu)\big) = T\big(x(\mu)\big)$, or, equivalently,

$$\mu'(i) = \arg \min_{u \in U(i)} \left[c_i(u) + \sum_{j=2}^{n} p_{ij}(u) x_j(\mu) \right], \qquad i = 2, 3, \ldots, n.$$

The new policy is strictly better if the current policy is nonoptimal; indeed, it was shown by Eq. (3.12) and the discussion following that equation that $x(\mu') \leq x(\mu)$, with strict inequality $x_i(\mu') < x_i(\mu)$ for at least one state x_i, if the policy $\{\mu, \mu, \ldots\}$ is nonoptimal. Because the number of stationary policies is finite, it follows that this policy iteration algorithm terminates after a finite number of iterations with an optimal proper policy. Note that each iteration involves a "policy evaluation" step, whereby, given $\mu \in M$, we obtain the corresponding cost vector $x(\mu)$ by solving the system of

equations $x(\mu) = c(\mu) + P(\mu)x(\mu)$ subject to the constraint $x_1(\mu) = 0$. This step can be very time–consuming when the number of states is large. If a powerful parallel machine is available, however, the policy evaluation step can be speeded up a great deal [to as little as $O(n)$ as shown in Section 2.2, or $O(\log^2 n)$ as shown in Section 2.9]. Under these circumstances, the policy iteration algorithm can be attractive, since it typically requires very few iterations for termination. We note also that a similar finitely terminating policy iteration algorithm can be shown to yield an optimal stationary policy under the Discounted Cost Assumption 3.1 starting from an arbitrary stationary policy [Ber87].

4.3.3 Parallel Implementation of the Dynamic Programming Iteration

We now consider the implementation of the dynamic programming (D.P.) iteration

$$x_i := \min_{u \in U(i)} \left[c_i(u) + \alpha \sum_{j=1}^{n} p_{ij}(u) x_j \right]$$

in a message–passing system [cf. Eq. (3.7)]. The situation here is similar as for the matrix–vector type of calculations discussed in Subsection 1.3.6. The main difference is the presence of the minimization over $u \in U(i)$. Indeed when $U(i)$ consists of a single element, the D.P. iteration consists of a matrix–vector multiplication followed by a vector addition [cf. Eq. (3.5) in Subsection 1.3.6].

We first assume that there are p processors, with p less than or equal to the number of states n, and for simplicity, we assume that n is divisible by p. It is then natural to let the jth processor update the cost of states $(j-1)k+1$ through jk, where $k = n/p$. Let us assume that each set $U(i)$ contains r elements, $p_{ij}(u)$ is nonzero for all u [so the matrix $P(u)$ is dense], and the jth processor holds the vector x and the values $c_i(u)$, and $p_{im}(u)$ for all $u \in U(i)$, m, and $i = (j-1)k+1$ through $i = jk$. Then the updating of the corresponding coordinates of x according to the D.P. iteration takes $O(knr)$ time for each processor. To communicate the results of the updating to the other processors, a multinode broadcast of packets, each containing k numbers, is necessary. If a linear array is used for communication, then we see, using the results of Subsection 1.3.4, that the multinode broadcast takes $O(kp) = O(n)$ time. Thus, the total time per iteration is $O(knr) = O(n^2 r/p)$.

Assume now that a hypercube with n^2 processors is available. We assume that n is a power of 2, and that the processors are arranged in an $n \times n$ array, with each "row" of n processors being itself a hypercube (cf. the algorithm for matrix–vector multiplication of Fig. 1.3.26 in Subsection 1.3.6). We also assume that at the beginning of each iteration, each processor (i, j) holds x_j and $p_{ij}(u)$ for all $u \in U(i)$. Furthermore, every processor (i, i) holds $c_i(u)$ for all $u \in U(i)$. Each D.P. iteration consists of four phases. In the first phase, each processor (i, j) forms the product $p_{ij}(u)x_j$ for all $u \in U(i)$; this takes $O(r)$ time. In the second phase, the sums $c_i(u) + \sum_{j=1}^{n} p_{ij}(u)x_j$, for all $u \in U(i)$, are

formed at node (i,i) by using r single node accumulations along the ith row hypercube; these can be pipelined (cf. Exercise 3.19 in Section 1.3) so that they take $O(r+\log n)$ time. In the third phase, each processor (i,i) computes the new value of x_i, which is the minimum over $u \in U(i)$ of the sums $c_i(u)+\sum_{j=1}^n p_{ij}(u)x_j$ over all $u \in U(i)$; this takes $O(r)$ time. Finally, in the fourth phase, each processor (i,i) broadcasts the new value of x_i to all processors (j,i), $j=1,2,\ldots,n$ along the ith column hypercube, requiring $O(\log n)$ time. The total time taken by the D.P. iteration is $O(r+\log n)$.

Suppose, finally, that a hypercube with $n^2 r$ processors is available. We assume that both r and n are powers of 2, and that the processors are arranged in an $n \times n \times r$ array, with each "row" of n or r processors being itself a hypercube. We number the r elements of $U(i)$ as $1,2,\ldots,r$, and we assume that at the beginning of each iteration, each processor (i,j,u) holds x_j and $p_{ij}(u)$, and each processor (i,i,u) holds $c_i(u)$. Then the D.P. iteration can be implemented similarly as in the preceding paragraph, but with modifications to take advantage of the additional processors. One difference is that the first and second phases are parallelized over u, so that they take time $O(1)$ and $O(\log n)$, respectively; furthermore, the minimization over $u \in U(i)$ of the third phase yielding x_i is done in time $O(\log r)$ using a single node accumulation at node $(i,i,1)$. Finally, at the fourth phase, x_i is transmitted from processor $(i,i,1)$ to all processors (j,i,u) in time $O\big(\log(rn)\big)$ using a single node broadcast. If $r \le n$, the total time per D.P. iteration is $O(\log n)$.

Note that the bound on the order of time per D.P. iteration given above for each case of number of processors and interconnection network cannot be improved in the absence of additional problem structure, even if all communication is assumed instantaneous.

EXERCISES

3.1. Let Assumption 3.1 hold, $e=(1,1,\ldots,1)$, and $\epsilon>0$ be a given scalar:

(a) Show that if $\mu \in M$ is such that

$$T_\mu(x^*) \le T(x^*) + \epsilon e,$$

then

$$x(\mu) \le x^* + \frac{\epsilon}{1-\alpha} e.$$

(b) Suppose that x satisfies $|x_i - x_i^*| \le \epsilon$ for all i. Show that if $\mu \in M$ is such that $T_\mu(x) = T(x)$, then

$$x^* \le x(\mu) \le x^* + \frac{2\alpha\epsilon}{1-\alpha} e.$$

3.2. Assume that $\alpha = 1$, $p_{ii}(u) = 0$ for all $i \neq 1$ and $u \in U(i)$, and $p_{11}(u) = 1$ for all $u \in U(1)$. For an improper policy $\{\mu, \mu, \ldots\}$, consider the directed graph having nodes $1, 2, \ldots, n$, and an arc (i, j) for each pair of nodes i and j such that $p_{ij}\big(\mu(i)\big) > 0$. Define the length of a cycle $(i_1, i_2, \ldots, i_k, i_1)$ as the sum $c_{i_1}\big(\mu(i_1)\big) + \cdots + c_{i_k}\big(\mu(i_k)\big)$. Show that if all cycles have positive length, the policy yields infinite cost for at least one initial state (cf. Assumption 3.2).

3.3. Assume that $\alpha = 1$, $p_{11}(u) = 1$ and $c_1(u) = 0$ for all $u \in U(1)$, and, furthermore, all stationary policies are proper. Show that the mappings T and T_μ of Eqs. (3.5) and (3.6), respectively, are contraction mappings with respect to some weighted maximum norm $\|\cdot\|_\infty^w$ over the subspace

$$X = \{x \in \Re^n \mid x_1 = 0\}.$$

Abbreviated proof. Partition the state space as follows. Let $S_1 = \{1\}$ and for $k = 2, 3, \ldots$, define sequentially

$$S_k = \left\{ i \;\middle|\; i \notin S_1 \cup \cdots \cup S_{k-1} \text{ and } \min_{u \in U(i)} \max_{j \in S_1 \cup \cdots \cup S_{k-1}} p_{ij}(u) > 0 \right\}.$$

Let S_m be the last of these sets that is nonempty. We claim that the sets S_k cover the entire state space, that is, $\cup_{k=1}^m S_k = S$. To see this, suppose that the set $S_\infty = \big\{ i \mid i \notin \cup_{k=1}^m S_k \big\}$ is nonempty. Then for each $i \in S_\infty$, there exists some $u_i \in U(i)$ such that $p_{ij}(u_i) = 0$ for all $j \notin S_\infty$. Take any μ such that $\mu(i) = u_i$ for all $i \in S_\infty$. The stationary policy $\{\mu, \mu, \ldots\}$ satisfies $\big[P^t(\mu)\big]_{ij} = 0$ for all $i \in S_\infty$, $j \notin S_\infty$, and t, and, therefore, cannot be proper, which contradicts the hypothesis.

We will choose a vector $w > 0$ so that T is a contraction with respect to $\|\cdot\|_\infty^w$ on the set X. We will take the ith coordinate w_i to be the same for states i in the same set S_k. In particular, we will choose the coordinates w_i of the vector w by

$$w_i = y_k \qquad \text{if} \qquad i \in S_k,$$

where $y_1, \ldots, y_m$ are appropriately chosen scalars satisfying

$$1 = y_1 < y_2 < \cdots < y_m. \tag{3.18}$$

Let

$$\epsilon = \min_{k=2,\ldots,m} \min_{\mu \in M} \min_{i \in S_k} \sum_{j \in S_1 \cup \cdots \cup S_{k-1}} \big[P(\mu)\big]_{ij}, \tag{3.19}$$

and note that $0 < \epsilon \leq 1$. We will show that it is sufficient to choose $y_2, \ldots, y_m$ so that for some $\gamma < 1$, we have

$$\frac{y_m}{y_k}(1 - \epsilon) + \frac{y_{k-1}}{y_k}\epsilon \leq \gamma < 1, \qquad k = 2, \ldots, m, \tag{3.20}$$

and then show that such a choice of $y_2, \ldots, y_m$ exists.

Indeed, for $x, x' \in X$, let $\mu \in M$ be such that $T_\mu(x) = T(x)$. Then we have for all i,

$$[T(x') - T(x)]_i = [T(x') - T_\mu(x)]_i \leq [T_\mu(x') - T_\mu(x)]_i = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)(x'_j - x_j). \quad (3.21)$$

Let $k(j)$ be such that j belongs to the set $S_{k(j)}$. Then we have, for any constant c,

$$\|x' - x\|_\infty^w \leq c \qquad \Rightarrow \qquad x'_j - x_j \leq c y_{k(j)}, \qquad j = 2, \ldots, n,$$

and Eq. (3.21) implies that for all i,

$$\begin{aligned}
\frac{\big[T(x')\big]_i - \big[T(x)\big]_i}{c y_{k(i)}} &\leq \frac{1}{y_{k(i)}} \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big) y_{k(j)} \\
&\leq \frac{y_{k(i)-1}}{y_{k(i)}} \sum_{j \in S_1 \cup \cdots \cup S_{k(i)-1}} p_{ij}\big(\mu(i)\big) + \frac{y_m}{y_{k(i)}} \sum_{j \in S_{k(i)} \cup \cdots \cup S_m} p_{ij}\big(\mu(i)\big) \\
&= \left(\frac{y_{k(i)-1}}{y_{k(i)}} - \frac{y_m}{y_{k(i)}} \right) \sum_{j \in S_1 \cup \cdots \cup S_{k(i)-1}} p_{ij}\big(\mu(i)\big) + \frac{y_m}{y_{k(i)}} \\
&\leq \left(\frac{y_{k(i)-1}}{y_{k(i)}} - \frac{y_m}{y_{k(i)}} \right) \epsilon + \frac{y_m}{y_{k(i)}} \leq \gamma,
\end{aligned}$$

where the second inequality follows from Eq. (3.18), the third inequality uses Eq. (3.19) and the fact $y_{k(i)-1} - y_m \leq 0$, and the last inequality follows from (3.20). Thus, we have

$$\frac{\big[T(x')\big]_i - [T(x)]_i}{w_i} \leq c\gamma, \qquad i = 1, \ldots, n,$$

so we obtain

$$\max_i \frac{\big[T(x')\big]_i - [T(x)]_i}{w_i} \leq c\gamma,$$

or

$$\big\|T(x) - T(x')\big\|_\infty^w \leq c\gamma, \qquad \text{for all } x, x' \in X \text{ with } \|x - x'\|_\infty^w \leq c.$$

It follows that T is a contraction mapping with respect to $\|\cdot\|_\infty^w$ over X.

We now show how to choose the scalars $y_1, y_2, \ldots, y_m$ so that Eqs. (3.18) and (3.20) hold. Let $y_0 = 0$, $y_1 = 1$, and suppose that $y_1, y_2, \ldots, y_k$ have been chosen. If $\epsilon = 1$, we choose $y_{k+1} = y_k + 1$. If $\epsilon < 1$, we choose y_{k+1} to be

$$y_{k+1} = \frac{1}{2}(y_k + M_k),$$

where

$$M_k = \min_{1\le i\le k} \left[y_i + \frac{\epsilon}{1-\epsilon}(y_i - y_{i-1}) \right].$$

Using the fact

$$M_{k+1} = \min \left\{ M_k, y_{k+1} + \frac{\epsilon}{1-\epsilon}(y_{k+1} - y_k) \right\},$$

it is seen by induction that for all k,

$$y_k < y_{k+1} < M_{k+1}.$$

In particular, we have

$$y_m < M_m = \min_{1\le i\le m} \left[y_i + \frac{\epsilon}{1-\epsilon}(y_i - y_{i-1}) \right],$$

which implies Eq. (3.20).

3.4. Prove Prop. 3.3 under a variation of Assumption 3.2, whereby, instead of assuming that every improper policy yields infinite cost for some initial state, we assume that $c_i(u) \ge 0$ for all i and $u \in U(i)$, and that there exists an optimal proper policy. The set X is now defined as $X = [x | x \ge 0, x_1 = 0]$. *Hint:* Lemma 3.1 is not valid, so a somewhat different argument is needed. The assumptions guarantee that x^* is finite and $x^* \in X$. [We have $x^* \ge 0$ because $c_i(u) \ge 0$, and $x_i^* < \infty$ because a proper policy exists.] The idea now is to show that $x^* \ge T(x^*)$, and then to choose μ such that $T_\mu(x^*) = T(x^*)$ and show that $\{\mu, \mu, \ldots\}$ is optimal and proper. Let $\pi = \{\mu_0, \mu_1, \ldots\}$ be a policy. We have for all i,

$$\big[x(\pi)\big]_i = c_i\big(\mu_0(i)\big) + \sum_{j=1}^{n} p_{ij}\big(\mu_0(i)\big)\big[x(\pi_1)\big]_j$$

where π_1 is the policy $\{\mu_1, \mu_2, \ldots\}$. Since $x(\pi_1) \ge x^*$, we obtain

$$\big[x(\pi)\big]_i \ge c_i\big(\mu_0(i)\big) + \sum_{j=1}^{n} p_{ij}\big(\mu_0(i)\big)x_j^* = \big[T_{\mu_0}(x^*)\big]_i \ge \big[T(x^*)\big]_i.$$

Taking the infimum over π in the preceding equation, we obtain

$$x^* \ge T(x^*). \tag{3.22}$$

Let $\mu \in M$ be such that $T_\mu(x^*) = T(x^*)$. From Eq. (3.22), we have $x^* \ge T_\mu(x^*)$, and using the monotonicity of T_μ, we obtain

$$x^* \ge T_\mu^t(x^*) = P^t(\mu)x^* + \sum_{k=0}^{t-1} P^k(\mu)c(\mu) \ge \sum_{k=0}^{t-1} P^k(\mu)c(\mu), \qquad \forall\, t \ge 1. \tag{3.23}$$

By taking limit superior as $t \to \infty$, we obtain $x^* \geq x(\mu)$. Hence, $\{\mu, \mu, \ldots\}$ is an optimal proper policy, and $x^* = x(\mu)$. Since μ was selected so that $T_\mu(x^*) = T(x^*)$, we obtain, using $x^* = x(\mu)$ and $x(\mu) = T_\mu\big(x(\mu)\big)$, $x^* = T(x^*)$. For the rest of the proof, use the vector Δ similarly as in the proof of Prop. 3.3.

3.5. Under the Discounted Cost Assumption 3.1, show that if $x \in \Re^n$ is such that $T(x) \geq x$, then $x^* \geq x$. Use this fact to show that x^* solves the linear program

$$\begin{aligned} &\text{maximize} \quad \beta' x \\ &\text{subject to} \quad c_i(u) + \alpha \sum_{j=1}^{n} p_{ij}(u) x_j \geq x_i, \qquad i = 1, \ldots, n, \ u \in U(i), \end{aligned}$$

where β is a nonzero vector with nonnegative coordinates. Derive a similar result under the Undiscounted Cost Assumption 3.2.

3.6. **[Tse85]** Consider the linear program in $\Re^n$:

$$\begin{aligned} &\text{maximize} \quad \beta' x \\ &\text{subject to} \quad C_k x \leq d_k, \qquad k = 1, \ldots, m \end{aligned}$$

where β is a nonzero vector with nonnegative coordinates, d_k are given vectors, and C_k are square $n \times n$ matrices with positive elements on the diagonal and nonpositive elements off the diagonal. Assume that all matrices C_k are diagonally dominant. Use the result of Exercise 3.5 to transform the problem into a dynamic programming problem.

3.7. Under the Undiscounted Cost Assumption 3.2 show that the Gauss–Seidel algorithm based on T (as defined in Subsection 3.1.2 of Chapter 3), converges to x^* for any initial vector $x \in X$. *Hint:* Compare with Exercise 1.4 in Section 3.1.

3.8. **(The Blackmailer's Dilemma [Whi83].)** The analysis given for undiscounted problems in this section relies on the finiteness of the set $U(i)$ of available decisions at state i. This exercise shows that if $U(i)$ is not finite, then Prop. 3.3 need not hold even if all stationary policies are proper. In particular, the optimal cost may be $-\infty$ for some initial states, there may exist an optimal nonstationary policy but no optimal stationary policy, and the dynamic programming algorithm may converge to a wrong point for some initial conditions.

Consider a controlled Markov chain with two states, 1 and 2. State 1 is absorbing and cost–free. In state 2 we must choose a control u from the interval $(0, 1]$ and incur a cost $-u$; we then move to state 1 at no cost with probability u^β, where β is a fixed positive scalar, and we stay in state 2 with probability $1 - u^\beta$. (We may view here u as a demand made by a blackmailer, and state 2 as the situation where the victim complies. State 1 is the situation where the victim refuses to pay and denounces the blackmailer to the police. The problem of this exercise models the blackmailer's effort to maximize the total expected gain through balancing at each time the desire for a high demand u with a low probability u^β that the victim will not comply.)

The mapping T takes the form

$$\big[T(x)\big]_1 = 0, \qquad \big[T(x)\big]_2 = \inf_{u \in (0,1]} \big[-u + \big(1 - u^\beta\big) x_2\big].$$

Show that:

(a) All stationary policies are proper and yield finite cost.

(b) If $\beta > 1$, then the optimal cost x_2^* starting from state 2 is $-\infty$. Show also that if $\beta = 2$ then the nonstationary policy that chooses in state 2 the control $u_k = \gamma/(k+1)$ at time k is optimal provided $\gamma \in (0, 1/2]$. *Hint:* Use the fact that $\sum_{k=1}^{\infty} 1/k = \infty$, and also that the product $\prod_{k=1}^{\infty}\left(1 - \gamma^2/k^2\right)$ is greater than $1 - \gamma^2 \sum_{k=1}^{\infty}(1/k^2)$ and is therefore positive for $\gamma \in (0, 1/2]$.

(c) If $\beta \leq 1$ then $x_2^* = -1$, and for all x with $x_1 = 0$, $x_2 \geq -1$ we have $T(x) = x^*$. On the other hand we have $x = T(x)$ for all x with $x_1 = 0$, $x_2 < -1$.

NOTES AND SOURCES

4.1 The shortest path problem is discussed in nearly every book on combinatorial and network optimization, see, e.g., [Law76], [PaS82], and [Roc84]. For a literature survey, see [DeP84] and for an extensive computational study, see [DGK79]. The Bellman–Ford algorithm is derived in [Bel57] and [For56]. Our treatment of arbitrary initial conditions (Prop. 1.2 in particular), appears to be new. The distributed Bellman–Ford algorithm as well as a parallel form of Dijkstra's method with a separate processor assigned to each origin node have been used in routing algorithms for data networks, including the ARPANET (see [BeG87], [Eph86], [MRR80], and [ScS80]). For distributed shortest path algorithms, appropriate for message–passing systems and data networks, see [Gal82] and [AwG87]. The parallel implementation of the Floyd–Warshall algorithm of Exercise 1.7 is similar to implementations on array processors and systolic arrays (see [AtK84], [DNS81], and [Kun88]). The parallel solution of a number of graph problems using various processor interconnection networks is discussed in [DNS81]. Shortest path algorithms based on forward search (cf. Exercise 1.10) are commonly used in artificial intelligence applications and are related to some branch–and–bound methods for integer programming; see [Ber87] and [Pea84].

4.2 Treatments of dynamic programming can be found in many textbooks, including [Bel57], [Ber87], [BeS78], [HeS82], [KuV86], [Nem66], [Ros83b], [Whi82], and [Whi83].

4.3 Discounted Markovian decision problems have been exhaustively analyzed and a detailed account with references and additional computational methods can be found in [Ber87].

Special cases of undiscounted Markovian decision problems, called *stopping problems* or *first passage problems*, have been considered extensively in the literature (see [Ber87] for an account and references). The theory existing up to now assumes that state costs are either all nonnegative or all nonpositive. Our treatment extends the theory to the case where both positive and negative state costs are allowed. Because our assumptions generalize naturally the standard Connectivity and Positive Cycle assumptions of the shortest path problem, we refer to the undiscounted problem as the stochastic shortest path problem.

Contraction mappings in dynamic programming are treated in [Den67] and [BeS78]. The weighted maximum norm contraction result of Exercise 3.3 has been attributed to A. J. Hoffman. The monotonicity of the dynamic programming mapping is emphasized in [Ber77], [BeS78], and [VeP87]; it will play an important role in the asynchronous version of the Bellman–Ford algorithm discussed in Section 6.4.

The complexity analysis of [PaT87b] strongly suggests that in contrast with the shortest path problem, there do not exist any parallel algorithms for solving the dynamic programming problems of this section in time $O(\log^k n)$ for any integer k.