Dynamic Programming.S1
Sequencing Problems

Many operational problems in manufacturing, service and distribution require the sequencing of various types of activities or items. Examples include a production facility in which chassis must be sequenced through an assembly line, an express mail service where parcels and letters must be routed for delivery, and a utility company that must schedule repair work. In general, problems in this class are easily formulated as mathematical programs but, with a few exceptions owing to special structure, are difficult to solve. In this section, we introduce a robust dynamic programming formulation that can be used to tackle a number of such problems. In most cases, however, the size of the state space is an exponential function of the number of items being sequenced. In practical instances, the success of the DP approach may depend on our ability to reduce the number of states that must be explored in the search for the optimum. One way to do this is to impose precedence requirements on the items to be sequenced; a second way is to introduce the logic of branch and bound within a dynamic programming algorithm.

Single Machine Scheduling

As a prototype, consider the problem of sequencing a set of n jobs through a single machine that can work on only one job at a time. Once a job is started, it must be completed without preemption. The time required to process job j once the machine begins to work on it is $p(j)$ for $j = 1, \ldots, n$. The associated cost $c(j, t)$ is a function of its completion time t and can take a variety of forms, the simplest being

$$c(j,t) = a(j)t$$

where $a(j)$ is the cost per unit time for job j. As we saw in Section 9.1 on greedy algorithms, this form admits a very simple solution when the objective is to minimize the total completion cost of all the jobs. The optimum is obtained by computing the ratio $p(j)/a(j)$ for each job and then sequencing them in order of increasing values of this ratio. The job with the smallest ratio is processed first, the job with next smallest ratio is processed second, and so on until all jobs are completed. Ties may be broken arbitrarily.

A much more difficult problem results when each job j has a due date $b(j)$. The cost of a job is zero if it is completed before its due date but increases linearly if it is tardy.

$$c(j,t) = \begin{cases} 0 & \text{for } 0 \le t \le b(j) \\ a(j)(t - b(j)) & \text{for } t > b(j) \end{cases}$$

The goal of the optimization is to determine the sequence that has the smallest total cost. Table 21 gives the relevant parameters for a 4-job

instance. There are 4! = 24 possible solutions. For the solution (3, 1, 2, 4), the completion times are 7, 12, 21 and 31 respectively. Jobs 3 and 1 are completed before their due date so no cost is incurred. Job 2 is 11 days late resulting in a cost of $440 and job 4 is 14 days late resulting in a cost of $420. The total cost is therefore $860.

Table 21. Job parameters for a sequencing problem

Job j	Processing time $p(j)$	Due date $b(j)$	Cost per day $a(j)$
1	5	12	$80
2	9	10	40
3	7	10	100
4	10	17	30

In general, we write a sequence as a vector $(j_1, j_2, j_3, \ldots, j_n)$ which implies that job j_1 is processed first, job j_2 second and so on until the final job j_n. This vector is a permutation of the integers 1 through n and admits $n!$ possible sequences, a number that increases rapidly with n. In fact, it is not possible to find a polynomial function of n that provides a bound on how fast $n!$ grows.

The time at which a job is finished is determined by its place in the sequence. Job j_k is in position k and is not started until the previous $k - 1$ jobs finish processing. It ends at time $t(j_k)$, the sum of the processing times of the previous jobs plus its processing time $p(j_k)$.

$$t(j_k) = \sum_{i=1}^{k} p(j_i)$$

The cost associated with a particular sequence is the sum of the individual job costs as determined by their completion times. The objective function is then

$$z = \sum_{j=1}^{n} c(j, t(j)).$$

The goal is to minimize z.

To solve this problem with dynamic programming, we must first describe it as a sequential decision process. In this case, the description is

once again straightforward with the decisions corresponding to places in the sequence. Thus the decision at each step is a job number. The DP model is given in Table 22.

Table 22. General sequencing problem

Component	Description
State	To determine the state definition, consider the information necessary to specify the set of feasible decisions and to evaluate the cost associated with a decision. At a particular step in the sequence, a job is a feasible choice if it hasn't been chosen before. Thus the minimal information the state must provide is the set of jobs previously included in the sequence. This also is the information necessary to compute the time of completion of the job and hence the associated cost. To describe the state we need a vector with n components $\mathbf{s} = (s_1, s_2, \ldots, s_n)$, where $s_j = \begin{cases} 0 & \text{if job } j \text{ has not been included in the sequence} \\ 1 & \text{if job } j \text{ has already been included in the sequence} \end{cases}$
Initial state set	$\boldsymbol{I} = \{(0, 0, \ldots, 0)\}$ None of the jobs has been scheduled.
Final state set	$\boldsymbol{F} = \{(1, 1, \ldots, 1)\}$ All jobs are scheduled.
State space	The state vector is a binary vector with n components. Therefore, there are 2^n members of the state space representing all possible combinations. $\boldsymbol{S} = \{\mathbf{s} : s_j = 0 \text{ or } 1,\ j = 1, \ldots, n\}$
Decision	The decision vector has a single component that identifies the next job to be processed. $\mathbf{d} = (d)$, where d = the next job in the sequence
Feasible decision	The feasible decisions at a given state are the jobs not already chosen. $\boldsymbol{D}(\mathbf{s}) = \{j : s_j = 0, j = 1, \ldots, n\}$

Transition function	The transition function changes the state to reflect the inclusion of an additional job in the sequence. $\mathbf{s}' = T(\mathbf{s}, \mathbf{d})$, where $s'_d = 1$ and $s'_j = s_j$ for $j \neq d$
Decision objective	$z(\mathbf{s}, \mathbf{d}) = c(d, t)$, where $t = \sum_{j=1}^{n} s_j p(j) + p(d)$ The cost function is problem dependent. For the job sequencing problem with tardiness penalties we use the cost function defined above.
Path objective	Minimize $z(\mathbf{P}) = \sum_{\mathbf{s}\ \mathbf{S},\mathbf{d}\ \boldsymbol{D}(\mathbf{s})} z(\mathbf{s}, \mathbf{d}) + f(\mathbf{s}_f)$
Final value function	$f(\mathbf{s}) = 0$ for $\mathbf{s}$ $\boldsymbol{F}$

Example 10 – *Job Sequencing with Tardiness Penalties*

The decision network for the data given in Table 21 is depicted in Fig. 14. The state vectors are shown in parentheses adjacent to the nodes. Arcs represent transition from one state to the next and each has an associated cost (not shown). The solution is found by finding the shortest path through the network and is shown by the heavy lines in the figure. The optimum is the sequence used as an example above.

Although the shortest path problem on an acyclic network can be solved efficiently, the difficulty here is that there are an exponential number of states, 2^n. This means that the DP approach as given in Table 11 does not lead to an efficient solution procedure for most sequencing problems; that is, the amount of computations is not bounded by a polynomial function of n. Because of the large number of states, problems can be solved only for small values of n[1].

The state space is considerably reduced if an ordering between some jobs is imposed. For example, if one specifies that job 3 must precede job 1, the number of feasible states is reduced from 16 to 12. Each additional restriction reduces the number of states in some nonlinear fashion.

[1] As a point of reference, we solved a 10 job problem with 1024 states in about eight minutes on a 400 Mz Macintosh G3 using the Excel add-in.

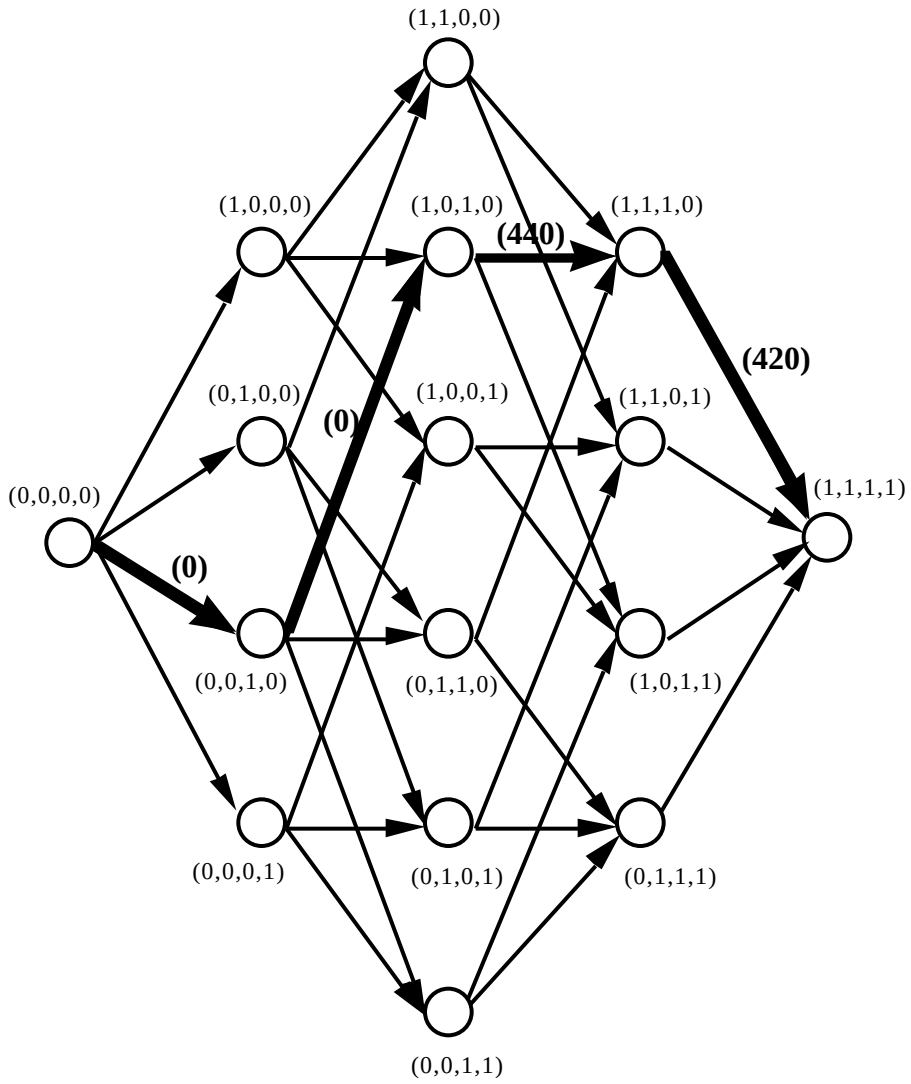

Figure 14. Decision network for 4-job sequencing problem

Traveling Salesman Problem

In our description of the total tardiness problem, the cost associated with a particular job did not depend on its immediate predecessor. There are many situations, though, where these costs are sequence dependent. In manufacturing, for example, it may be necessary to change the tooling between two successive jobs, or in scheduling propane deliveries, the length of the route and hence travel cost depends on the order in which customers are visited. In these cases, it would be necessary to extend the definition of the state space in Table 22 to include an additional state representing the last job processed in the sequence. The traveling salesman problem (TSP fits this situation.

Recall that in the TSP a salesman must visit n cities starting and ending at his home base. The objective is to minimize some measure of travel cost subject to the restriction that each city must be visited once and only once. A feasible solution is called a tour. We arbitrarily identify city 1 as the home base. Like the sequencing problem, a solution is described by a vector $(1, j_2, j_3, \ldots, j_n)$ which implies that the tour starts at city 1, goes next to city j_2, and so on until the final city j_n is reached. To complete the tour, the salesman must travel from j_n back to city 1. The cost of the tour is

$$z = c(1, j_2) + \sum_{k=2}^{n-1} c(j_k, j_{k+1}) + c(j_n, 1)$$

where the function $c(i, j)$ specifies the cost of traveling from city i to city j. When $c(i, j)$ represents the distance between i and j, the objective of the problem is simply to minimize the total distance traveled.

The dynamic programming model is similar to the sequencing model in that the state identifies the set of cities that have been visited at any point in the tour. To compute the cost of traveling to the next city, though, we need to the city visited. An additional state variable is defined for this purpose.

Table 23. Traveling salesman problem

Component	Description
State	$\mathbf{s} = (s_1, s_2, \ldots, s_n, s_{n+1})$, where $s_j = \begin{cases} 0 & \text{if city } j \text{ is not in the sequence} \\ 1 & \text{if city } j \text{ is in the sequence} \end{cases}$ $j = 1,\ldots,n$ s_{n+1} = index of the last city in the sequence
Initial state set	$\boldsymbol{I} = \{(1, 0, \ldots, 0, 1)\}$ Only city 1 is in the tour and that is the last city visited.
Final state set	$\boldsymbol{F} = \{(1, 1, \ldots, 1, j) : j = 2, \ldots, n\}$ All cities are in the tour. The last city can be any city but 1.

State space	There are 2^{n-1} possible combinations of the first n state variables, since s_1 is fixed as 1. The last state variable can take on $n-1$ values. $\boldsymbol{S} = \{\mathbf{s} : s_1 = 1,\ s_j = 0 \text{ or } 1,\ j = 2,\ldots,n \text{ and } s_{n+1} = 2,\ldots,n\}$
Decision	The decision vector has a single component that identifies the next city to be included in the tour. $\mathbf{d} = (d)$, where d = the next city in the tour
Feasible decision	$\boldsymbol{D}(\mathbf{s}) = \{j : s_j = 0, j = 2,\ldots,n\}$ The feasible decisions at a given state are the cities not yet visited.
Transition function	The transition function changes the state to reflect the inclusion of an additional city in the tour. The last state variable becomes the decision. $\mathbf{s}' = T(\mathbf{s}, \mathbf{d})$, where $s'_d = 1$, $s'_j = s_j$ for $j \neq d$, and $s_{n+1} = d$
Decision objective	$z(\mathbf{s}, \mathbf{d}) = c(s_{n+1}, d)$ where $c(\cdot,\cdot)$ is defined for all city pairs.
Path objective	Minimize $z(\mathbf{P}) = \sum_{\mathbf{s}\ \boldsymbol{S},\mathbf{d}\ \boldsymbol{D}(\mathbf{s})} z(\mathbf{s}, \mathbf{d}) + f(\mathbf{s}_f)$
Final value function	$f(\mathbf{s}) = c(s_{n+1}, 1)$ for $\mathbf{s}$ $\boldsymbol{F}$ This function is the cost of traveling from the last city to city 1.

Example 11 – *Traveling Salesman Problem*

Consider an eight city problem on a square grid with the coordinates assigned randomly in the range 0 to 25. The following matrix shows the locations of the cities in the (x,y)-plane.

	x	*y*
1	7.0	9.2
2	20.0	9.3
3	20.6	15.3

4	9.0	7.5
5	6.6	13.7
6	4.2	5.2
7	4.3	4.7
8	13.9	12.7

For the cost function we use the *p*-norm distance between a city pair given by

$$c(i,j) = \left(|x_i - x_j|^p + |y_i - y_j|^p \right)^{1/p} .$$

When p =2, this function gives the Euclidean distance between the two points; when p = 1, the function gives the rectilinear distance. Other values are possible. For the example, we used p = 2.

The dynamic programming model of the problem has 449 states. The optimal solution is shown in Fig. 15. The effort required to solve the problem is primarily influenced by the number of states. It is possible to reduce this number if precedence relations can be specified between city pairs.

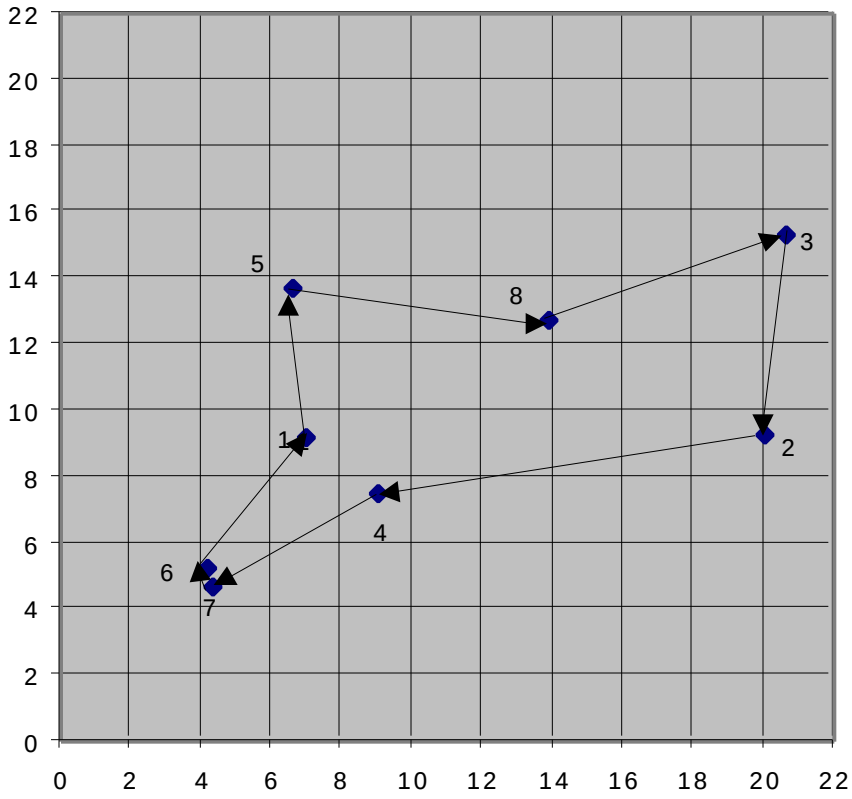

Figure 15. Optimal solution to TSP