VPP SLIDES – v7.3

11.11.2025

Course Information

- Lectures: general theory, small exercises, optional attendance

- Computer-assisted seminars: larger examples, MATLAB, compulsory attendance
  - each week you'll be given exercises for the next week's seminar – you are expected to read them before the seminar
  - you can try and finish the exercises by yourself – if you succeed and send me the correct solutions, you can skip the credit project

- Credit: active participation in the seminars, elaboration of a given project
  - either correct solution to more than half of the exercises
  - or video on a given topic (more on that later)

- Exam: Written and oral

- Prerequisites: mathematical analysis, algebra, statistics and probability

Sources:

- Bertsekas, D. P.: Dynamic Programming and Optimal Control (Volume I & II). Athena Scientific, 2007.

- Puterman, M. L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley-Interscience, 2005.

- Brucker, P.: Scheduling Algorithms. Springer, 2010.

- Pinedo, M. L.: Scheduling: Theory, Algorithms, and Systems. Springer, 2016.

- Martí, R., Pardalos P. M., Resende M. G. C.: Handbook of Heuristics. Springer, 2025.

- Luke, S.: Essentials of Metaheuristics. Lulu.com, 2012.

- Google disk (link): slides, videos, exercises, MATLAB codes, ...

Topics: (order might change, we may skip some)

- Introduction to dynamic programming

- Deterministic problems and shortest paths

- Branch and Bound algorithm

- Constrained and multiobjective problems

- Project management: CPM & PERT

- Deterministic continuous-time problems

- LQR and Kalman filter

- Approximate DP and Model predictive control

- Infinite time problems

- Scheduling

1   Introduction

- almost all of the material is based on the book by Bertsekas, Dynamic Programming and Optimal Control, 3rd edition, 2005

- we deal with situations where decisions are made in **stages**

- the outcome of each decision may not be fully predictable but can be anticipated to some extent before the next decision is made

- the objective is to minimize a certain cost

- decisions cannot be viewed in isolation since one must balance the desire for low present cost with the undesirability of high future costs

- our basic model has two principal features: (1) an underlying **discrete dynamic system**, and (2) a **cost function that is additive over time** (extensions exist)

The system has the form

$$x_{k+1} = f_k(x_k, u_k, w_k), \quad k = 0, 1, \ldots, N-1,$$

where

k indexes discrete time,

x_k is the state of the system and summarizes past information that is relevant for the future optimization,

u_k the control or decision variable to be selected at time k,

w_k is a random parameter (also called disturbance or noise depending on the context),

N is the horizon or number of times control is applied,

and f_k is a function that describes the system and in particular the mechanism by which the state is updated.

- the cost function is additive in the sense that the cost incurred at time k, denoted by $g_k(x_k, u_k, w_k)$, accumulates over time

- total cost:

$$g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k),$$

where $g_N(x_N)$ is a terminal cost incurred at the end. Because of the random variable w_k, we need a more sensible cost:

$$E\left\{ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k) \right\}$$

- the optimization is over the controls $u_0, u_1, \ldots, u_{N-1}$, but some qualification is needed here; each control u_k is selected with some knowledge of the current state x_k (current value or related information).

Example 1.1 – Inventory Control

Consider a problem of ordering a quantity of certain item at each of N periods so as to (roughly) meet a stochastic demand, while minimizing the incurred expected cost. Let us denote

x_k stock available at the beginning of the kth period

u_k stock ordered (and immediately delivered) at the beginning of the kth period

w_k demand during the kth period with given probability distribution

We assume that $w_0, w_1, \ldots, w_{N-1}$ are independent random variables, and that excess demand is backlogged and filled as soon as additional inventory becomes available.

- system equation:

$$x_{k+1} = x_k + u_k - w_k$$

- cost has two components:
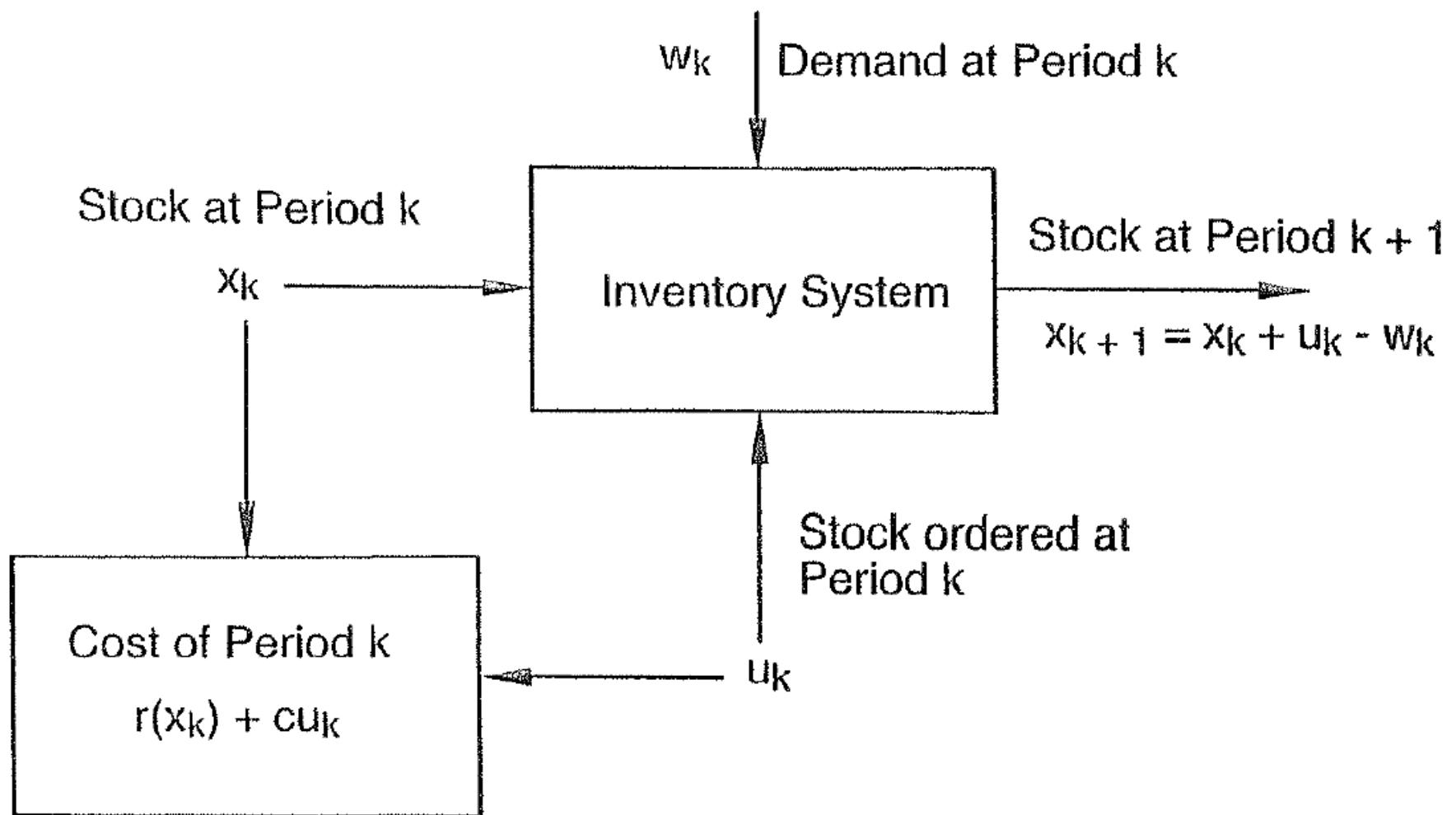
  $r(x_k)$ – representing penalty for either positive stock (holding/inventory cost) or negative stock (unfulfilled demand)

  cu_k – purchasing cost, where c is cost per unit ordered

- terminal cost $R(x_N)$ for being left with inventory x_N at the end

- total cost:

$$E\left\{ R(x_N) + \sum_{k=0}^{N-1} (r(x_k) + cu_k) \right\}$$

w_k | Demand at Period k

Stock at Period k

x_k ——— Inventory System

Stock at Period k + 1

$x_{k+1} = x_k + u_k - w_k$

Stock ordered at Period k

u_k

Cost of Period k

$r(x_k) + cu_k$

open-loop vs. closed-loop minimization:

- open-loop: select all orders $u_0, \ldots, u_{N-1}$ at once at time 0, without waiting to see the subsequent demand levels

- closed-loop: postpone placing the order u_k until the last possible moment (time k) when the current stock x_k will be known

- in particular, in closed-loop inventory optimization we are not merely interested in finding optimal numerical values of the orders but rather we want to find an **optimal rule for selecting at each period k an order u_k for each possible value of stock x_k that can conceivably occur**

- mathematically, we want to find a **sequence of functions** μ_k, $k = 0, \ldots, N-1$, mapping stocks x_k into order u_k as to minimize the expected cost:
  $$\mu_k(x_k) = \text{ amount that should be ordered at time } k \text{ if the stock is } x_k$$

- the sequence $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ will be referred to as a **policy** or **control law**

Discrete-State and Finite-State Problems

- in the preceding example, the state x_k was a continuous real variable, and it is easy to think of multidimensional generalizations where the state is an n-dimensional vector of real variables

- it is also possible that the state takes values from a discrete set, such as the integers

- a version of the inventory problem where a discrete viewpoint is more natural arises when stock is measured in whole units (such as cars), each of which is a significant fraction of x_k, u_k or w_k

- there are many situations where the state is naturally discrete and there is no continuous counterpart – such situations are conveniently specified in terms of the probabilities of transitions between the states

- $p_{ij}(u, k)$ is the probability at time k that the next state will be j, given that the current state is i, and the control selected is u:

$$p_{ij}(u, k) = P\{x_{k+1} = j \mid x_k = i, u_k = u\}$$

- this type of state transition can alternatively be described in terms of the discrete-time equation $x_{k+1} = w_k$, where the probability of the random parameter w_k is $P\{w_k = j \mid x_k = i, u_k = u\} = p_{ij}(u, k)$

- conversely, given a discrete-state system in the form $x_{k+1} = f_k(x_k, u_k, w_k)$, together with the probability distribution $P_k(w_k \mid x_k, u_k)$ of w_k, we can provide an equivalent transition probability description

- the corresponding probabilities are given by

$$p_{ij}(u, k) = P_k\{W_k(i, u, j) \mid x_k = i, u_k = u\},$$

where $W_k(i, u, j) = \{w \mid j = f_k(i, u, w)\}$

Basic Problem

- general problem of decision under uncertainty over a finite number of stages

- a discrete-time system:

$$x_{k+1} = f_k(x_k, u_k, w_k), \quad k = 0, 1, \ldots, N-1,$$

  where $x_k \in S_k, u_k \in C_k, w_k \in D_k$. The control u_k is constrained to take values in a given nonempty subset $U(x_k) \subset C_k$, which depends on the current state x_k, that is $u_k \in U_k(x_k)$ for all $x_k \in S_k$ and k

- the random disturbance w_k is characterized by a probability distribution $P_k(\,\cdot\,|\,x_k, u_k)$ that may depend explicitly on x_k and u_k but not on values of prior disturbances $w_{k-1}, \ldots, w_0$ (Markov property)

- we consider the class of policies (or control laws) that consist of sequence of functions

$$\pi = \{\mu_0, \ldots, \mu_{N-1}\},$$

where μ_k maps states x_k into controls $u_k = \mu_k(x_k)$ and is such that $\mu_k(x_k) \in U_k(x_k)$ for all $x_k \in S_k$. Such policies are called **admissible**

- given an initial state x_0 and an admissible policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$, the states x_k and disturbances w_k are random variables with distribution defined through:

$$x_{k+1} = f_k(x_k, \mu_k(x_k), w_k), \quad k = 0, 1, \ldots, N-1$$

- for a given function g_k, $k = 1, \ldots, N$, the expected cost of π starting at x_0 is

$$J_\pi(x_0) = E\left\{ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \right\}$$

- an optimal policy π^* is one that minimizes this cost, that is

$$J_{\pi^*}(x_0) = \min_{\pi \in \Pi} J_\pi(x_0),$$
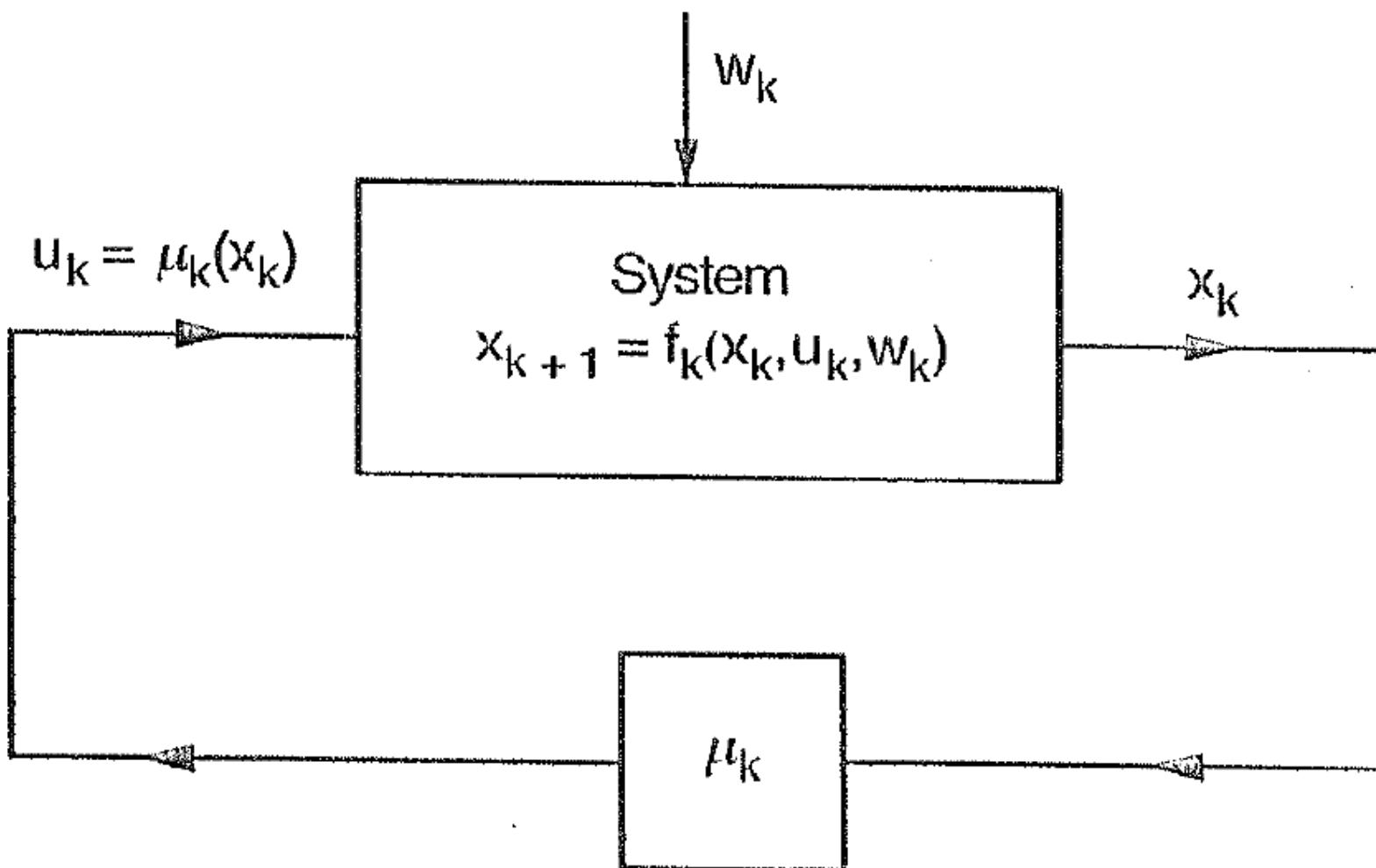
where Π is a set of all admissible policies. The optimal cost depends on x_0 and is denoted by $J^*(x_0)$, that is

$$J^*(x_0) = \min_{\pi \in \Pi} J_\pi(x_0).$$

- it is useful to view J^* as a function that assigns to each initial state x_0 the optimal cost $J^*(x_0)$ and call it the optimal cost function or optimal value function

The Role and Value of Information

- open-loop vs. closed-loop minimization

- with closed-loop policies, it is possible to achieve lower cost, essentially by taking advantage of the extra information (the value of the current state)

- this reduction in cost may be called the **value of the information** and can be rather significant

- if the information is not available, the controller cannot adapt appropriately to the unexpected values of the state

- in the inventory control example, the information that becomes available at the beginning of each period k is the inventory stock x_k – clearly a very important information w.r.t. selecting u_k

$$w_k$$

$$u_k = \mu_k(x_k)$$

System

$$x_{k+1} = f_k(x_k, u_k, w_k)$$

$$x_k$$

$$\mu_k$$

The Dynamic Programming Algorithm

- the dynamic programming (DP) technique rests on a very simple idea, the **principle of optimality**, due to R. Bellman

- **Principle of optimality:** Let $\pi^* = \{\mu_0^*, \mu_1^*, \dots, \mu_{N-1}^*\}$ be an optimal policy for the basic problem, and assume that when using π^*, a given state x_i occurs at time i with positive probability. Consider the subproblem whereby we are at x_i at time i and wish to minimize the "cost-to-go" from time i to time N
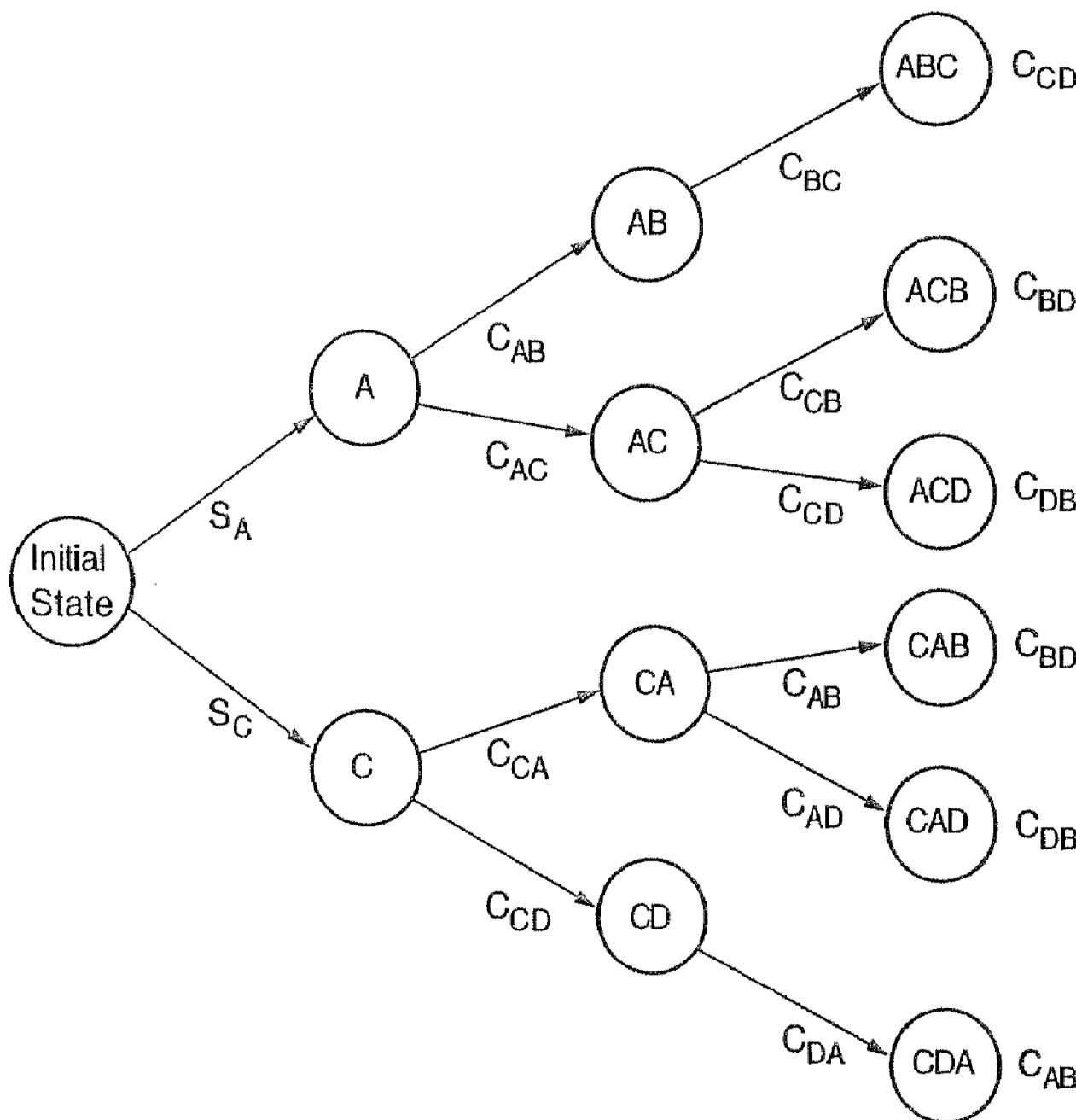
$$
E\left\{ g_N(x_N) + \sum_{k=i}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \right\}
$$

  Then the truncated policy $\{\mu_i^*, \mu_{i+1}^*, \dots, \mu_{N-1}^*\}$ is optimal for this subproblem.

- intuitive justification: if the truncated policy $\{\mu_i^*, \mu_{i+1}^*, \ldots, \mu_{N-1}^*\}$ were not optimal as stated, we would be able to reduce the cost further by switching to an optimal policy from for the subproblem once we reach x_i

- the principle of optimality suggests that an optimal policy can be constructed in a piecemeal fashion, first constructing an optimal policy for the "tail subproblem" involving the last stage, then extending the optimal policy to the "tail subproblem" involving the last two stages, and continuing in this manner until an optimal policy is constructed

- the DP algorithm is based on this idea: it proceeds sequentially, by solving all the tail subproblems of a given time length, using the solution of the tail subproblems of shorter time length.

Example 1.2 – Deterministic Scheduling

- suppose that to produce a certain product, four operations must be performed on a certain machine – A, B, C, and D

- assume that operation B can be performed only after operation A has been performed, and operation D only after operation C (i.e. CDAB is ok, CDBA is not)

- the setup cost C_{mn} for passing from any operation m to any other operation n is given. There is also an initial startup cost S_A or S_C for starting with operations A or C. Cost of ACBD is: $S_A + C_{AC} + C_{CB} + C_{BD}$

- we can view this problem as a sequence of three decisions (the fourth operation is uniquely determined)

- deterministic problem with a finite number of states can be represented by a transition graph.

S_A	5
S_C	3
C_{AB}	2
C_{AC}	3
C_{AD}	4
C_{BC}	3
C_{BD}	1
C_{CA}	4
C_{CB}	4
C_{CD}	6
C_{DA}	3
C_{DB}	3

- now we state the DP algorithm for the basic problem and show its optimality:

Proposition 1: For every initial state x_0, the optimal cost $J^*(x_0)$ of the basic problem is equal to $J_0(x_0)$, given by the last step of the following algorithm, which proceeds backward in time from period $N - 1$ to period 0:

$$J_N(x_N) = g_N(x_N),$$

$$J_k(x_k) = \min_{u_k \in U_k(x_k)} E_{w_k} \left\{ g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \right\}, \quad k = 0, \ldots, N-1,$$

where the expectation is w.r.t. the probability distribution of w_k, which depends on x_k and u_k. Furthermore, if $u_k^* = \mu_k^*(x_k)$ minimizes the right side of the equation above for each x_k and k, the policy $\pi^* = \{\mu_0^*, \mu_1^*, \ldots, \mu_{N-1}^*\}$ is optimal.

---

* the proof of this proposition will be somewhat informal and assumes that the functions J_k are well-defined and finite – for strictly rigorous proof see the Bertsekas' book [Section 1.5]

Proof*:[1/2] For any admissible policy $\pi = \{\mu_0, \mu_1, \ldots, \mu_{N-1}\}$ and each $k = 0, 1, \ldots, N-1$, denote $\pi^k = \{\mu_k, \mu_{k+1}, \ldots, \mu_{N-1}\}$. For $k = 0, 1, \ldots, N-1$, let $J_k^*(x_k)$ be the optimal cost for the $(N-k)$-stage problem that start at state x_k and time k, and ends at time N,

$$J_k^*(x_k) = \min_{\pi^k} E_{w_k, \ldots, w_{N-1}} \left\{ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, u_i, w_i) \right\}.$$

For $k = N$, we define $J_N^*(x_N) = g_N(x_N)$. We will show, by induction that the functions J_k^* are equal to the functions J_k generated by the DP algorithm, so that for $k = 0$, we will obtain the desired result.

Indeed, we have by definition $J_N^* = J_N = g_N$. Assume that for some k and all x_{k+1}, we have $J_{k+1}^*(x_{k+1}) = J_{k+1}(x_{k+1})$. Then, since $\pi^k = (\mu_k, \pi^{k+1})$, we have for all x_k

$$J_k^*(x_k) = \min_{(\mu_k, \pi^{k+1})} E_{w_k, \ldots, w_{N-1}} \left\{ g_k(x_k, \mu_k(x_k), w_k) + g_N(x_N) + \sum_{i=k+1}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right\}$$

Proof[*]:[2/2]

$$J_k^*(x_k) = \min_{(\mu_k, \pi^{k+1})} E_{w_k, \ldots, w_{N-1}} \left\{ g_k(x_k, \mu_k(x_k), w_k) + g_N(x_N) + \sum_{i=k+1}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right\}$$

$$= \min_{\mu_k} E_{w_k} \left\{ g_k(x_k, \mu_k(x_k), w_k) + \right.$$

$$\left. \min_{\pi^{k+1}} \left[E_{w_{k+1}, \ldots, w_{N-1}} \left\{ g_N(x_N) + \sum_{i=k+1}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right\} \right] \right\}$$

$$= \min_{\mu_k} E_{w_k} \left\{ g_k(x_k, \mu_k(x_k), w_k) + J_{k+1}^*(f_k(x_k, \mu_k(x_k), w_k)) \right\}$$

$$= \min_{\mu_k} E_{w_k} \left\{ g_k(x_k, \mu_k(x_k), w_k) + J_{k+1}(f_k(x_k, \mu_k(x_k), w_k)) \right\}$$

$$= \min_{u_k \in U_k(x_k)} E_{w_k} \left\{ g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \right\}$$

$$= J_k(x_k),$$

completing the induction. (the conversion of the minimization over μ_k to a minimization over u_k uses the fact that for any function F of x and u, we have
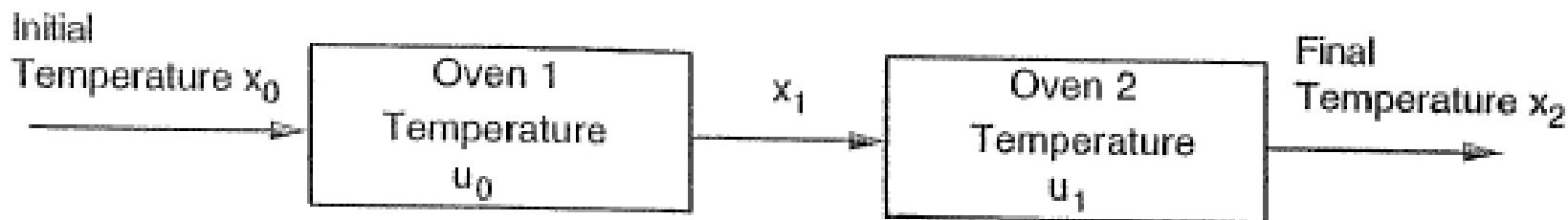
$$\min_{\mu \in M} F(x, \mu(x)) = \min_{u \in U(x)} F(x, u),$$

where M is the set of all functions $\mu(x)$ such that $\mu(x) \in U(x)$ for all x.)

- the argument of the preceding proof provides an interpretation of $J_k(x_k)$ as the optimal cost for an $(N - k)$-stage problem starting at state x_k and time k, and ending at time N

- we consequently call $J_k(x_k)$ the **cost-to-go** at state x_k and time k, and refer to J_k as the **cost-to-go function** at time k

- ideally, we would like to use the DP algorithm to obtain closed-form expressions for J_k or an optimal policy

- unfortunately, in many practical cases an analytic solution is not possible, and one has to resort to numerical executions of the DP algorithm

- the computational requirements are proportional to the number of possible values of x_k, so for complex problems the computational burden may become prohibitive (curse of dimensionality)

Example 1.3 – Deterministic Ovens (continuous states)

A certain material is passed through a sequence of two ovens

x_0: initial temperature of the material,

x_k: temperature of the material at the exit of oven $k = 1, 2$,

u_{k-1}: prevailing temperature in oven $k = 1, 2$.

System model:

$$x_{k+1} = (1 - a)x_k + au_k, \quad k = 0, 1.$$

where a is a known scalar from the interval (0,1).

The objective is to get the final temperature x_2 close to a given target T, while expending relatively little energy. This is expressed by a cost function

$$r(x_2 - T)^2 + u_0^2 + u_1^2,$$

where $r > 0$ is a given scalar. We assume no constraints on u_k. We have $N = 2$ and a terminal cost $g_2(x_2) = r(x_2 - T)^2$, so the initial condition of the DP algorithm is

$$J_2(x_2) = r(x_2 - T)^2.$$

For the next-to-last stage, we have

$$J_1(x_1) = \min_{u_1}[u_1^2 + J_2(x_2)] = \min_{u_1}[u_1^2 + J_2((1-a)x_1 + au_1)].$$

Substituting the previous form of J_2, we get

$$J_1(x_1) = \min_{u_1}[u_1^2 + r((1-a)x_1 + au_1 - T)^2].$$

Unconstrained minimization over u_1 – simple derivatives (for constrained KKT conditions).

$$0 = 2u_1 + 2ra((1 - a)x_1 + au_1 - T),$$

and by solving for u_1 we get the optimal temperature for the last oven:

$$\mu_1^*(x_1) = \frac{ra(T - (1 - a)x_1)}{1 + ra^2}.$$

Note that this is not a single control but rather a control function, a rule that tells us the optimal oven temperature $u_1 = \mu_1^*(x_1)$ for each possible state x_1. By substituting the optimal u_1 to J_1, we get

$$J_1(x_1) = \frac{r^2a^2((1 - a)x_1 - T)^2}{(1 + ra^2)^2} + r\left((1 - a)x_1 + \frac{ra^2(T - (1 - a)x_1)}{1 + ra^2} - T\right)^2$$

$$J_1(x_1) = \frac{r^2a^2((1 - a)x_1 - T)^2}{(1 + ra^2)^2} + r\left(\frac{ra^2}{1 + ra^2} - 1\right)^2((1 - a)x_1 - T)^2$$

$$J_1(x_1) = \frac{r((1 - a)x_1 - T)^2}{1 + ra^2}.$$

We now go back one more stage:

$$J_0(x_0) = \min_{u_0}[u_0^2 + J_1(x_1)] = \min_{u_0}[u_0^2 + J_1((1 - a)x_0 + au_0)],$$

by substituting expression obtained for J_1, we get

$$J_0(x_0) = \min_{u_0}\left[u_0^2 + \frac{r((1 - a)^2 x_0 + (1 - a)au_0 - T)^2}{1 + ra^2}\right].$$

Again, minimization by setting the derivative equal to zero:

$$0 = 2u_0 + \frac{2r(1 - a)a((1 - a)^2 x_0 + (1 - a)au_0 - T)}{1 + ra^2}.$$

This yields, after some calculation, the optimal temperature of the first oven:

$$\mu_0^*(x_0) = \frac{r(1 - a)a(T - (1 - a)^2 x_0)}{1 + ra^2(1 + (1 - a)^2)}.$$

The optimal cost is obtained by substituting this expression in the formula for J_0. This leads to a straightforward but lengthy computation, which yields the rather simple formula

$$J_0(x_0) = \frac{r((1 - a)^2 x_0 - T)^2}{1 + ra^2(1 + (1 - a)^2)}.$$

Why did it work so well? Linear system equation and quadratic costs.

Example 1.4 – Resource Allocation

- we are asked to allocate an amount b of some resource between N activities

- system state: x_k – how much of the resource is left to allocate for the kth activity

- decision: u_k – how much to allocate for the kth activity

- system equation and constraints:

$$x_{k+1} = x_k - u_k, \quad k = 0, \ldots, N-1,$$

  where $u_k \in [0, x_k]$, $x_0 = b$, $x_k \in [0, b]$

- cost functions: $g_k(u_k)$ – positive effect from allocating amount u_k of the resource to activity k

- overall effect: $g_N(x_N) + \sum_{k=0}^{N-1} g_k(u_k)$ (which we want to maximize)

- example: x – money in your wallet, $N = 2$ activities (buying something nice to eat and drink), $b = 1000$ (amount of money to spend) and

$$g_0(u_0) = 25\sqrt{\max(0, u_0 - 100)} \qquad \text{(effect of buying drinks)}$$

$$g_1(u_1) = 20\sqrt{u_1} \qquad \text{(effect of buying food)}$$

$$g_2(x_2) = x_2 \qquad \text{(money remaining)}$$

- $N = 2$:

$$J_2(x_2) = g_2(x_2) = x_2$$

- $N = 1$:

$$J_1(x_1) = \max_{u_1 \in [0, x_1]} g_1(u_1) + J_2(x_1 - u_1)$$

$$= \max_{u_1 \in [0, x_1]} 20\sqrt{u_1} + (x_1 - u_1)$$

$$\frac{d}{du_1}(20\sqrt{u_1} + (x_1 - u_1)) = \frac{10}{\sqrt{u_1}} - 1$$

$\rightarrow u_1 = \mu_1(x_1) = \min(x_1, 100), \;\; J_1(x_1) = 20\sqrt{\min(x_1, 100)} + x_1 - \min(x_1, 100)$

$$J_1(x_1) = \begin{cases} x_1 + 100, & x_1 \geq 100 \\ 20\sqrt{x_1}, & x_1 < 100 \end{cases}$$

- $N = 0$:

$$J_0(x_0) = \max_{u_0 \in [0,x_0]} g_0(u_0) + J_1(x_0 - u_0)$$

$$= \max_{u_0 \in [0,x_0]} 25\sqrt{\max(0, u_0 - 100)} + 20\sqrt{\min(x_0 - u_0, 100)}$$

$$+ x_0 - u_0 - \min(x_0 - u_0, 100)$$

- we will find the optimum with the help of numerical methods (it is possible to find it analytically, but the effort is not worth it)

- second possibility: discretize the problem into finitely many states – we can spend money only in whole amounts, i.e. $u_0 \in [0, 1, \ldots, x_0], u_1 \in [0, 1, \ldots, x_1]$, $x_{0,1,2} \in [0, 1, \ldots, b]$ – straightforward to implement/program

Example 1.5 – Knapsack Problem

- one of the most well known NP-hard problems

- there are N items we can put inside a knapsack (backpack) that has capacity V

- every item k has a corresponding value c_k and weight v_k

- our job is to optimally select the items to take in order to maximize the value of the knapsack

- there are many variants and extensions: multidimensional, multicriteria, travelling thief problem, etc.

- if the weights are integers, the problem is solvable by the DP algorithm

- the stages correspond to the individual items (artificial time)

- systems state: x_k - how much space remains in the backpack before we choose to take the kth item

- $x_0 = V$, x_N - how much space is left in the backpack after going through all the items

- decision: $u_k \in \{0, 1\}$ - don't/do take the kth item, if $x_k < v_k$, we are forced to select $u_k = 0$

- system equation: $x_{k+1} = x_k - v_k u_k, \quad k = 0, \ldots, N - 1$

- value functions for each item: $g_k(x_k, u_k) = c_k u_k, \quad k = 0, \ldots, N - 1$

- value function for the last stage: $g_N(x_N) = 0$ (no value for the empty space)

- DP algorithm ($\mathcal{O}(WN)$ vs. $\mathcal{O}(2^N)$ brute-force):

$$J_N(x_N) = 0,$$

$$J_k(x_k) = \max_{u_k} c_k u_k + J_{k+1}(x_k - v_k u_k), \quad k = 0, \ldots, N-1.$$
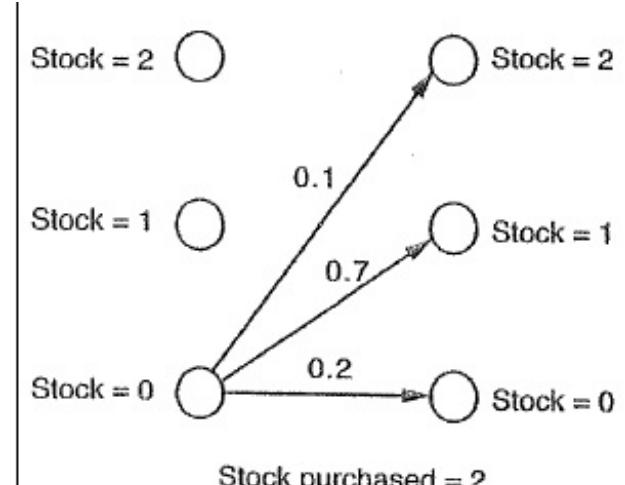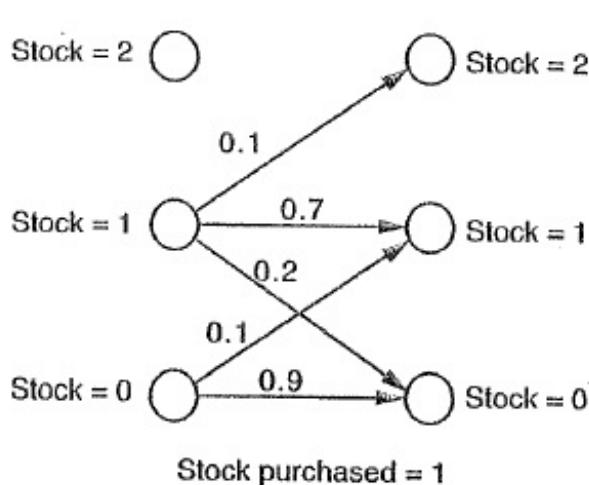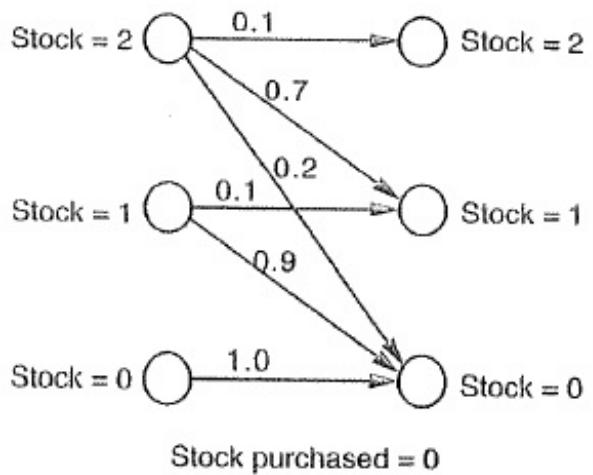
item k	0	1	2	3
c_k	11	12	8	6
v_k	4	6	3	2
V	8			

x_k	J_0	J_1	J_2	J_3	J_4
0					
1					
2					
3					
4					
5					
6					
7					
8					

x_k	μ_0	μ_1	μ_2	μ_3
0				
1				
2				
3				
4				
5				
6				
7				
8				

Example 1.6 – Inventory Control (revisited)

- similar setting as in Example 1.1, but the inventory x_k and the demand w_k are nonnegative integers and the excess demand $(w_k - x_k - u_k)$ is lost

- system equation: $x_{k+1} = \max(0, x_k + u_k - w_k)$

- constraint on stocked units: $x_k + u_k \leq 2$

- storage cost for the kth period: $(x_k + u_k - w_k)^2$

- ordering cost is 1 per unit ordered

- cost per period: $g_k(x_k, u_k, w_k) = u_k + (x_k + u_k - w_k)^2$

- terminal cost assumed to be 0: $g_N(x_N) = 0$

- planning horizon $N = 3$, with $x_0 = 0$

- the demand has the same distribution for all stages:
$$p(w_k = 0) = 0.1, \quad p(w_k = 1) = 0.7, \quad p(w_k = 2) = 0.2$$

Stock purchased = 0   Stock purchased = 1   Stock purchased = 2

- stage/period $k = 3$:

$$J_3(x_3) = g_3(x_3) = 0$$

- $k = 2, 1, 0$:

$$J_k(x_k) = \min_{\substack{0 \le u_k \le 2 - x_k \\ u_k = 0,1,2}} E_{w_k} \left\{ u_k + (x_k + u_k - w_k)^2 + J_{k+1}(\max(0, x_k + u_k - w_k)) \right\}$$

x_k	J_0	μ_0	J_1	μ_1	J_2	μ_2	J_3
0							0
1							0
2							0

- $k = 2$:
    - $x = 0$:

$$J_2(0) = \min_{u_2=0,1,2} E_{w_2}\left\{u_2 + (0 + u_2 - w_2)^2 + 0\right\}$$

$$= \min_{u_2=0,1,2}\left[u_2 + 0.1(u_2)^2 + 0.7(u_2 - 1)^2 + 0.2(u_2 - 2)^2\right]$$

$$J_2(0) = 1.3, \qquad \mu_2(0) = 1$$

    - $x = 1$:

$$J_2(1) = \min_{u_2=0,1} E_{w_2}\left\{u_2 + (1 + u_2 - w_2)^2 + 0\right\}$$

$$= \min_{u_2=0,1}\left[u_2 + 0.1(1 + u_2)^2 + 0.7(u_2)^2 + 0.2(u_2 - 1)^2\right]$$

$$J_2(1) = 0.3, \qquad \mu_2(1) = 0$$

    - $x = 2$:

$$J_2(2) = \min_{u_2=0} E_{w_2}\left\{u_2 + (2 + u_2 - w_2)^2 + 0\right\}$$

$$= \min_{u_2=0}\left[u_2 + 0.1(2 + u_2)^2 + 0.7(1 + u_2)^2 + 0.2(u_2)^2\right]$$

$$J_2(2) = 1.1, \qquad \mu_2(2) = 0$$

x_k	J_0	μ_0	J_1	μ_1	J_2	μ_2	J_3
0					1.3	1	0
1					0.3	0	0
2					1.1	0	0

- $k = 1$:

  - $x = 0$:
$$J_1(0) = \min_{u_1=0,1,2} E_{w_1} \left\{ u_1 + (0 + u_1 - w_1)^2 + J_2(\max(0, 0 + u_1 - w_1)) \right\}$$

$$u_1 = 0 : E\{\cdot\} = 0.1 \cdot J_2(0) + 0.7(1 + J_2(0)) + 0.2(4 + J_2(0)) = 2.8$$

$$u_1 = 1 : E\{\cdot\} = 1 + 0.1(1 + J_2(1)) + 0.7 \cdot J_2(0) + 0.2(1 + J_2(0)) = 2.5$$

$$u_1 = 2 : E\{\cdot\} = 2 + 0.1(4 + J_2(2)) + 0.7(1 + J_2(1)) + 0.2 \cdot J_2(0) = 3.68$$

$$J_1(0) = 2.5, \qquad \mu_1(0) = 1$$

  - $\vdots$

Minimax (Robust) Control

- the problem of optimal control of uncertain systems has traditionally been treated in stochastic framework – all uncertain quantities are described by probability distributions, and the expected value of the cost is minimized

- in many practical situations a stochastic description of the uncertainty may not be available

- a less detailed structure is sometimes available, such as bounds on the magnitude of the uncertain quantities

- we may know a set within which the uncertain quantities are known to lie, but may not know the corresponding probability distribution

- under these circumstances, one may use a minimax approach, where the minimization is w.r.t. the worst-possible outcome, i.e. the objective is to

$$\text{minimize} \ \max_{w \in W} J(\pi, w),$$

over all $\pi \in \Pi$ (generally no closed form solution)

- in the framework of the basic problem, consider the case where the disturbances $w_0, w_1, \ldots, w_{N-1}$ are known to belong to corresponding given sets $W_k(x_k, u_k) \subset D_k, k = 0, 1, \ldots, N - 1$

- the problem is to find a policy $\pi = \{\mu_0, \mu_1, \ldots, \mu_{N-1}\}$ with $\mu_k(x_k) \in U_k(x_k)$ for all x_k and k, which minimizes the cost function

$$J_\pi(x_0) = \max_{\substack{w_k \in W_k(x_k, \mu_k(x_k)) \\ k=0,1,\ldots,N-1}} \left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \right]$$

- the DP algorithm for this problem has the following form (maximization instead of the expectation):

$$J_N(x_N) = g_N(x_N),$$

$$J_k(x_k) = \min_{u_k \in U_k(x_k)} \max_{w_k \in W_k(x_k, u_k)} [g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k))]$$

[proof in Section 1.6 of the Bertsekas' book]

State Augmentation and Reformulations

- what to do in situations, when some assumptions of the basic problem are violated

- generally, in such cases the problem can be reformulated into the basic problem format by **space augmentation** (typically by the enlargement of the state space)

- general guideline: include in the enlarged state at time k all the information that is known to the controller at time k and can be used with advantage in selecting u_k

- often has a price: larger and more complex state space

Time lags: in many applications the system state x_{k+1} depends not only on the preceding state x_k and control u_k, but also on earlier states and controls (i.e. some time lag)

- state augmentation: the state is expanded to include an appropriate number of earlier states and controls

- for a simple single period time lag in the state and control:
$$x_{k+1} = f_k(x_k, x_{k-1}, u_k, u_{k-1}, w_k), \quad k = 1, 2, \ldots, N-1, \quad x_1 = f_0(x_0, u_0, w_0),$$
if we introduce additional state variables y_k and s_k and make the identification $y_k = x_{k-1}, s_k = u_{k-1}$, the system equation yields
$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \\ s_{k+1} \end{pmatrix} = \begin{pmatrix} f_k(x_k, y_k, u_k, s_k, w_k) \\ x_k \\ u_k \end{pmatrix}$$
and by defining $\tilde{x}_k = (x_k, y_k, s_k)$ as the new state, we have
$$\tilde{x}_{k+1} = \tilde{f}_k(\tilde{x}_k, u_k, w_k)$$

- extending the state space in this fashion will have large consequences on the number of possible elements the state can have

- consider a finite-state situation with n possible states in x and m possible controls in u (both independent of k) - there are now $n \times n \times m$ possible values for the state (instead of just n):

	J_0	μ_0	J_1	μ_1	...
(x_k^1, y_k^1, s_k^1)					
...					
(x_k^i, y_k^j, s_k^l)					
$(x_k^i, y_k^{j+1}, s_k^l)$					
$(x_k^i, y_k^j, s_k^{l+1})$					
$(x_k^i, y_k^{j+1}, s_k^{l+1})$					
$(x_k^{i+1}, y_k^{j+1}, s_k^l)$					
$(x_k^{i+1}, y_k^j, s_k^{l+1})$					
...					
(x_k^n, y_k^n, s_k^m)					

Correlated disturbances: disturbances w_k are correlated over time. A common situation that can be handled efficiently by state augmentation arises when the process $w_0, \ldots, w_{N-1}$ can be represented as the output of a linear system driven by independent random variables

- example: by some appropriate statistical methods, we determined that the evolution of w_k can be modeled by an equation

$$w_k = \lambda w_{k-1} + \xi_k,$$

  where λ is a given scalar and $\{\xi_k\}$ is a sequence of independent random vectors with given distribution

- state augmentation: we can introduce an additional state variable

$$y_k = w_{k-1}$$

and obtain a new system equation

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} f_k(x_k, u_k, \lambda y_k + \xi_k) \\ \lambda y_k + \xi_k \end{pmatrix}$$

the new state is $\tilde{x}_k = (x_k, y_k)$ and the new disturbance vector is ξ_k

- more generally, suppose that w_k can be modelled as

$$w_k = C_k y_{k+1}$$

  where

$$y_{k+1} = A_k y_k + \xi_k, \quad k = 0, \ldots, N - 1,$$

  A_k, C_k are matrices of appropriate dimensions and ξ_k are independent random vectors with given distributions

- by viewing y_k as an additional state variable, we obtain the new system equation

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} f_k(x_k, u_k, C_k(A_k y_k + \xi_k)) \\ A_k y_k + \xi_k \end{pmatrix}$$

- note, however, that in order to have a perfect state information, the controller must be able to observe y_k (unfortunately, this is true only in the minority of practical cases)

Simplification of Uncontrollable State Components

- when augmenting the state of a given system one often ends up with composite states, consisting of several components

- if some of these components cannot be affected by the choice of control, the DP algorithm can be simplified considerably

- let the state of the system be a composite (x_k, y_k) of two components x_k and y_k. The evolution of the main component, x_k, is affected by the control u_k according to the equation
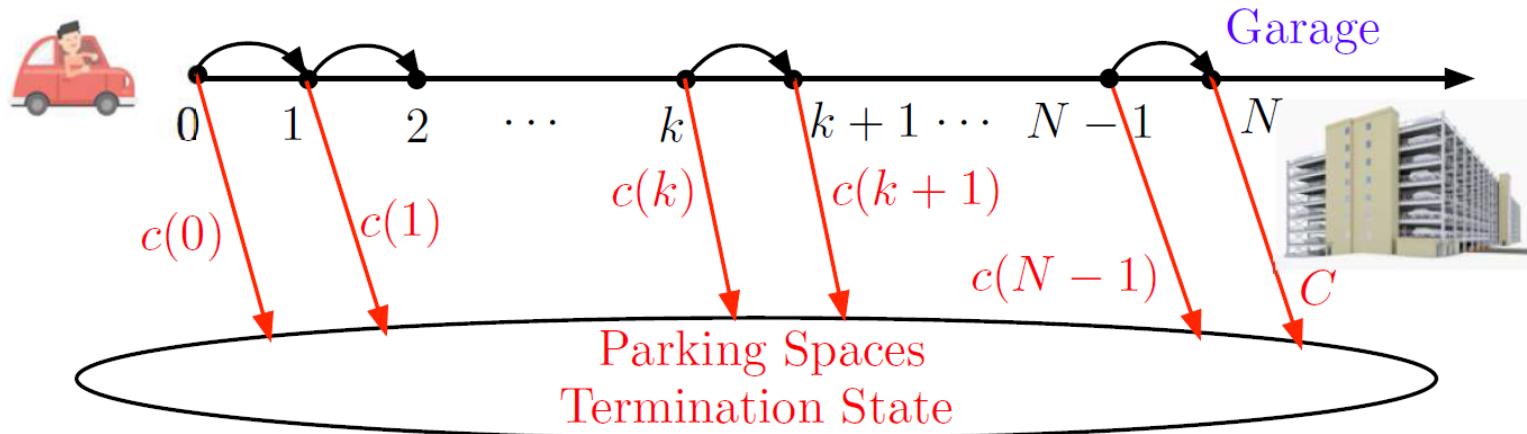
$$x_{k+1} = f_k(x_k, y_k, u_k, w_k),$$

  where the probability distribution $P_k(w_k \mid x_k, y_k, u_k)$ is given.

- the evolution of y_k is governed by a given conditional distribution $P_k(y_k \mid x_k)$ and cannot be affected by the control, except indirectly through x_k

Example 1.7 – Parking

- a driver is looking for inexpensive parking on the way to his destination, the parking area contains N spaces, and a garage at the end

- the driver starts at space 0 and traverses the parking spaces sequentially, i.e., from space k he goes next to space $k + 1$, etc.

- each parking space k costs $c(k)$ and is free with probability $p(k)$ independently of whether other parking spaces are free or not

- if the driver reaches the last parking space and does not park there, he must park at the garage, which costs C

- the driver can observe whether a parking space is free only when he reaches it, and then, if it is free, he makes a decision to park in that space or not to park and check the next space

- the problem is to find the minimum expected cost parking policy

Garage

$$0 \quad 1 \quad 2 \quad \cdots \quad k \quad k+1 \cdots \quad N-1 \quad N$$

$c(k)$ $c(k+1)$

$c(0)$ $c(1)$

$c(N-1)$ C

Parking Spaces
Termination State

- we formulate the problem as a DP problem with N stages, corresponding to the parking spaces, and an artificial terminal state t that corresponds to having parked

- at each stage $k = 0, \ldots, N-1$, in addition to t, we have two states (k, F) and $(k, \bar{F})$, corresponding to space k being free or taken, respectively

- the decision/control is to park or continue at state (k, F) (there is no choice at states $(k, \bar{F})$ and the garage)

- let us denote:

  $J_k^*(F)$: The optimal cost-to-go upon arrival at a space k that is free.

  $J_k^*(\bar{F})$: The optimal cost-to-go upon arrival at a space k that is taken.

$J_N^*(t) = C$: The cost-to-go upon arrival at the garage.

 $J_k^*(t) = 0$: The terminal cost-to-go.

- the DP algorithm for $k = 0, \ldots, N-1$ takes the form

$$J_k^*(F) = \begin{cases} \min[c(k), p(k+1)J_{k+1}^*(F) + (1 - p(k+1))J_{k+1}^*(\bar{F})] & \text{if } k < N-1, \\ \min[c(N-1), C] & \text{if } k = N-1, \end{cases}$$

$$J_k^*(\bar{F}) = \begin{cases} p(k+1)J_{k+1}^*(F) + (1 - p(k+1))J_{k+1}^*(\bar{F}) & \text{if } k < N-1, \\ C & \text{if } k = N-1, \end{cases}$$

(we omit here the obvious equations for the terminal state t and the garage state N)

- while this algorithm is easily executed, it can be written in a simpler and equivalent form, which takes advantage of the fact that the second component (F or $\bar{F}$) of the state is uncontrollable

- this can be done by introducing the scalars

$$\hat{J}_k = p(k)J_k^*(F) + (1 - p(k))J_k^*(\bar{F}), \quad k = 0, \ldots, N - 1,$$

which can be viewed as the optimal expected cost-to-go upon arriving at space k but **before verifying its free or taken status**

- indeed, from the preceding DP algorithm, we get

$$\hat{J}_{N-1} = p(N-1)\min[c(N-1), C] + (1 - p(N-1))C,$$

$$\hat{J}_k = p(k)\min[c(k), \hat{J}_{k+1}] + (1 - p(N-1))\hat{J}_{k+1}, \quad k = 0, \ldots, N - 2$$

- from this algorithm we can also obtain the optimal parking policy, which is to park at space $k = 0, \ldots, N - 1$ if it is free and $c(k) \leq \hat{J}_{k+1}$

- now, we will formulate a DP algorithm that is executed over the controllable component of the state, with the dependence on the uncontrollable component being "averaged out"

- let $J_k(x_k, y_k)$ denote the optimal cost-to-go at stage k and state (x_k, y_k), and define
$$\hat{J}_k(x_k) = E_{y_k}\{J_k(x_k, y_k) \mid x_k\},$$

the derivation of the DP algorithm that generates $\hat{J}_k(x_k)$:

$$
\begin{aligned}
\hat{J}_k(x_k) &= E_{y_k}\{J_k(x_k, y_k) \mid x_k\} \\
&= E_{y_k}\Big\{ \min_{u_k \in U_k(x_k, y_k)} E_{w_k, x_{k+1}, y_{k+1}}\{g_k(x_k, y_k, u_k, w_k) \\
&\qquad + J_{k+1}(x_{k+1}, y_{k+1}) \mid x_k, y_k, u_k\} \mid x_k\Big\} \\
&= E_{y_k}\Big\{ \min_{u_k \in U_k(x_k, y_k)} E_{w_k, x_{k+1}}\{g_k(x_k, y_k, u_k, w_k) \\
&\qquad + E_{y_{k+1}}\{J_{k+1}(x_{k+1}, y_{k+1}) \mid x_{k+1}\} \mid x_k, y_k, u_k\} \mid x_k\Big\} \\
&= E_{y_k}\Big\{ \min_{u_k \in U_k(x_k, y_k)} E_{w_k}\{g_k(x_k, y_k, u_k, w_k) \\
&\qquad + \hat{J}_{k+1}\left(f_k(x_k, y_k, u_k, w_k)\right)\} \mid x_k\Big\}
\end{aligned}
$$

- the advantage of this equivalent DP algorithm is that it is executed over a significantly reduced state space

- e.g., if x_k takes n possible values and y_k takes m possible values, then DP is executed over n states instead of nm states

- note that the minimization yields an optimal control law as a function of the full state space (x_k, y_k)

- uncontrolled state components often occur in arrival systems, such as queueing, where action must be taken in response to a random event (such as a customer arrival) that cannot be influenced by the choice of control

Finite-State Markov Chains

- very brief overview of the topic

- a square $n \times n$ matrix p_{ij} is called a stochastic matrix if all its elements are nonnegative ($p_{ij} \geq 0$) and the sum of the elements of each of its rows is equal to 1 ($\sum_{j=1}^{n} p_{ij} = 1$, for all $i = 1, \ldots, n$)

- we are given a stochastic $n \times n$ matrix P together with a finite set of states $S = \{1, \ldots, n\}$

- the pair (S, P) will be referred to as a stationary finite-state Markov chain

- we associate with (S, P) a process whereby an initial state $x_0 \in S$ is chosen in accordance with some initial probability distribution:

$$r_0 = (r_0^1, r_0^2, \ldots, r_0^n)$$

- a transition from state x_0 to a new state $x_1 \in S$ in accordance with a probability distribution P as follows: the probability that the new state will be j is equal to p_{ij} whenever the initial state is i, i.e.

$$P(x_1 = j \mid x_0 = i) = p_{ij}, \quad i, j = 1, \ldots, n$$

- similarly, subsequent transitions produce states $x_2, x_3, \ldots$

$$P(x_{k+1} = j \mid x_k = i) = p_{ij}, \quad i, j = 1, \ldots, n$$
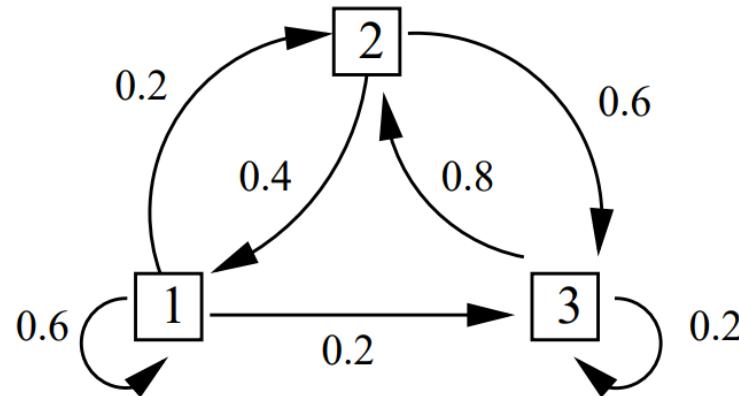
- these transition probabilities depend only on the current state k, not on all the states before that – Markov (memoryless) property

- the probability that after the kth transition the state x_k will be j, given that the initial state x_0 is i, is denoted by

$$p_{ij}^k = P(x_k = j \mid x_0 = i), \quad i, j = 1, \ldots, n$$

- a straightforward calculation shows that these probabilities are equal to the elements of the matrix $P^k = P \cdot P \cdots \cdot P$ (k times)

- the probability distribution of the state x_k after k transitions is

$$r_k = (r_k^1, r_k^2, \ldots, r_k^n) = r_0 P^k, \quad k = 1, 2, , \ldots$$

Example 1.8 – Simple Markov Chain

$$P \;=\; \begin{pmatrix} \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{pmatrix}, \quad r_0 = \left(0.5,\; 0.3,\; 0.2\right), \quad r_3 = \left(\ldots,\; \ldots,\; \ldots\right)$$

2   Deterministic Systems and the Shortest Path Problem

- deterministic problems – each disturbance w_k can take only one value

- such problems arise in many important contexts and also in situations where the problem is really stochastic but, as an approximation, the disturbance is fixed to some typical value

- important property: **using feedback results in no advantage in terms of cost reduction**, i.e. minimizing the cost over admissible policies $\{\mu_0, \ldots, \mu_{N-1}\}$ results in the same optimal cost as minimizing over sequences of control vectors $\{u_0, \ldots, u_{N-1}\}$

- this is true because given a policy $\{\mu_0, \ldots, \mu_{N-1}\}$ and the initial state x_0, the future states are perfectly predictable through the equation

$$x_{k+1} = f_k(x_k, \mu_k(x_k)), \quad k = 0, 1, \ldots, N-1,$$

  and the corresponding controls are perfectly predictable through
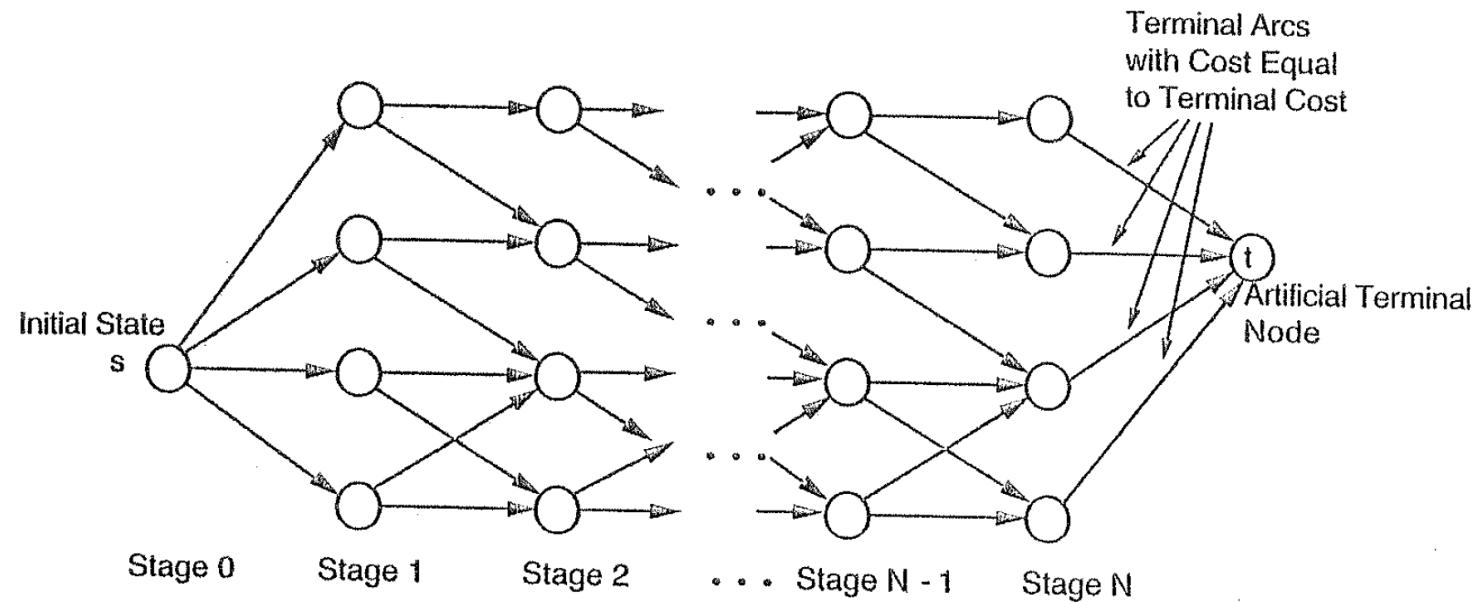
$$u_k = \mu_k(x_k), \quad k = 0, 1, \ldots, N-1$$

- this difference between deterministic and stochastic problems often has important computational implications – in deterministic problems with a "continuous space" character, optimal control sequences may be found by deterministic variational techniques, and by iterative optimal control algorithms (steepest descent, conjugate gradient, Newton's method, etc.)

- on the other hand, DP has a wider scope of applicability since it can handle difficult constraint sets such as integer or discrete sets

Finate-State Systems and Shortest Paths

- consider a deterministic problem where the state space S_k is a finite set for each k.

- at any state x_k, a control u_k can be associated with a transition from the state x_k to the state $f_k(x_k, u_k)$, at cost $g_k(x_k, u_k)$

- thus, finite-state deterministic problems can be equivalently represented by a graph, where the arcs correspond to transitions between states at successive stages and each arc has an associated cost

- to handle the final stage, an artificial terminal node t has been added – each state x_N at stage N is connected to the terminal node t with an arc having cost $g_N(x_N)$

- control sequences – paths from the initial state (node s at stage 0) and terminating at one of the nodes corresponding to the final stage N

- **deterministic finite-state problem is equivalent to finding a minimum-length (or shortest) path from the initial node s of the graph to the terminal node t**

- transition graph – nodes correspond to states, an arc with start at x_k and end in x_{k+1} corresponds to a transition of the form $x_{k+1} = f_k(x_k, u_k)$, the cost of the transition is the length of the arc

- here, a path means a sequence of arcs of the form $(j_1, j_2), (j_2, j_3), \ldots, (j_{k-1}, j_k)$, and by the length of a path we mean the sum of the lengths of its arcs

- a slight change of notation:

$$a_{i,j}^k = \text{ cost of transition at stage } k \text{ from state } i \in S_k \text{ to state } j \in S_{k+1}$$

$$a_{i,t}^N = \text{ terminal cost of state } i \in S_N \text{ [which is } g_N(i)\text{]},$$

where we adopt the convention $a_{ij}^k = \infty$ if there is no control that moves the state from i to j at stage k

- the DP algorithm takes the form

$$J_N(i) = a_{it}^N, \quad i \in S_N,$$

$$J_k(i) = \min_{j \in S_{k+1}} [a_{ij}^k + J_{k+1}(j)], \quad i \in S_k, \quad k = 0, 1, \dots, N-1$$

- the optimal cost is $J_0(s)$ and is equal to the length of the shortest path from s to t

A Forward DP Algorithm

- the preceding algorithm proceeds **backward** in time, but it is possible to derive an equivalent algorithm that proceeds **forward** in time (but just for this special case of deterministic finite-state problems!)

- simple observation: an optimal path from s to t is also an optimal path from t to s in a "reverse" shortest path problem where the direction of each arc is reversed and its length is left unchanged

- the DP algorithm for this "reverse" problem starts from states $x_1 \in S_1$ of stage 1, proceeds to states $x_2 \in S_2$ of stage 2, and continues all the way to states $x_N \in S_N$:

$$\tilde{J}_N(j) = a_{sj}^0, \quad j \in S_1,$$

$$\tilde{J}_k(j) = \min_{i \in S_{N-k}} [a_{ij}^{N-k} + \tilde{J}_{k+1}(i)], \quad j \in S_{N-k+1}, \quad k = 1, 2, \ldots, N-1,$$

with the optimal cost

$$\tilde{J}_0(t) = \min_{i \in S_N} [a_{it}^N + \tilde{J}_1(i)]$$

- the backward and forward algorithms yield the same result in the sense that
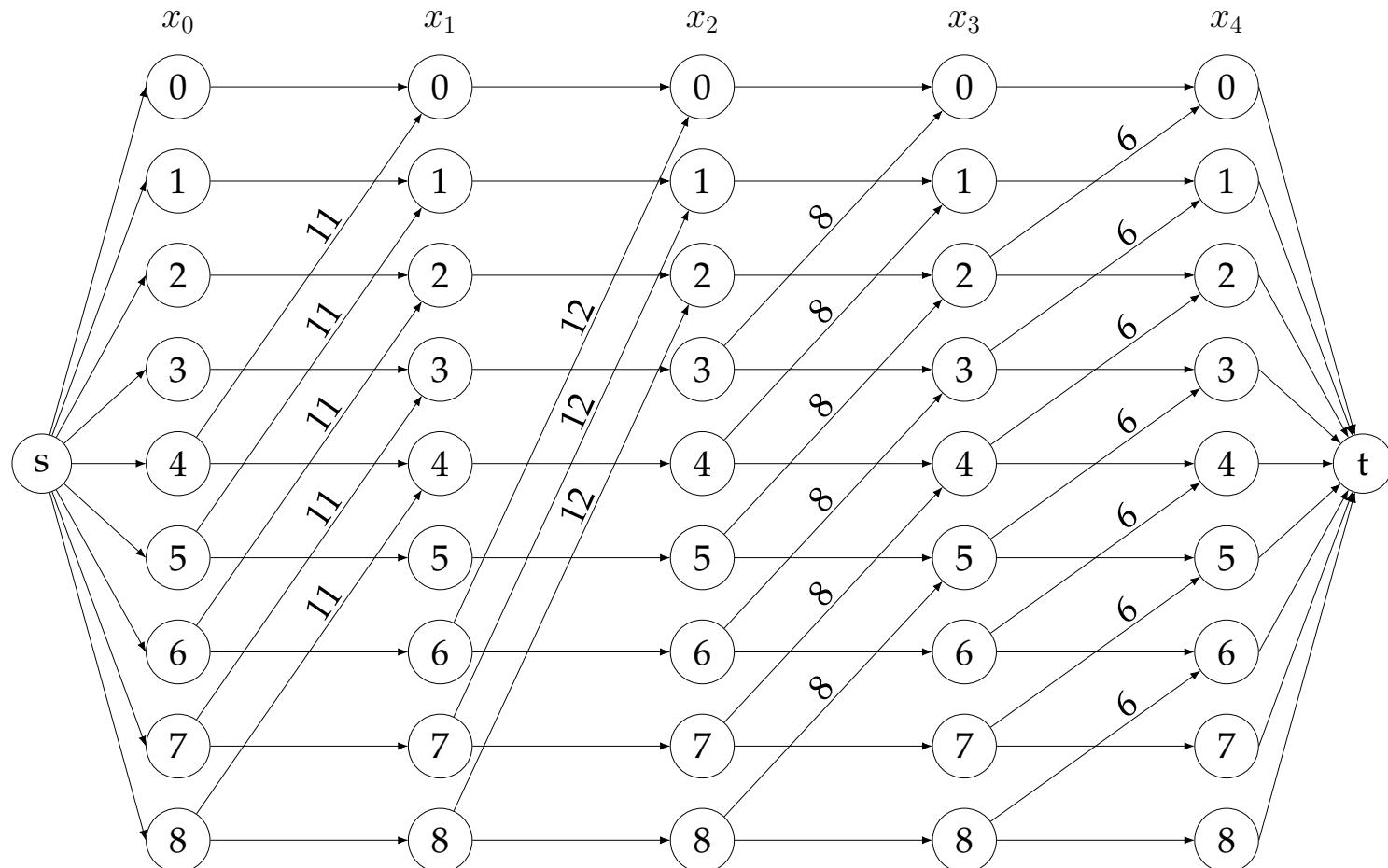
$$J_0(s) = \tilde{J}_0(t)$$

and an optimal control sequence (or shortest path) obtained from any one of the two is optimal for the original problem

- we may view $\tilde{J}_k(j)$ as an optimal **cost-to-arrive** to state j from the initial state s, in contrast with $J_k(i)$, which represents the optimal cost-to-go from state i to the terminal state t

- important use of the forward DP algorithm is in real-time applications where the stage k problem data are unknown prior to stage k and are revealed to the controller just before stage k begins

- also, any shortest path problem can be posed as a deterministic finite-state DP problem

Example 2.1 – Knapsack (revisited)

item k	0	1	2	3
c_k	11	12	8	6
w_k	4	6	3	2
W	8			

Converting a Shortest Path Problem to a Deterministic Finite-State Problem

- Let $\{1, 2, \ldots, N, t\}$ be a set of nodes of a graph, and let a_{ij} be the cost of moving from node i to node j (length of the arc joining i and j)

- node t is a special node – the **destination**

- $a_{ij} = \infty$ when there is no arc joining nodes i and j

- we want to find a shortest path from each node i to node t, i.e. a sequence of moves that minimizes total cost to get to t from each of the nodes $1, 2, \ldots, N$

- important assumption: circles, i.e. paths of the form $(i, j_1), (j_1, j_2), \ldots, (j_k, i)$, cannot have negative total length

- with this assumption, it is clear that an optimal path need not take more than N moves, so we may limit the number of moves to N

- we formulate the problem as one where we require exactly N moves but allow degenerate moves from a node i to itself with cost $a_{ii} = 0$

- denote for $i = 1, \ldots, N, k = 0, \ldots, N-1$,

$$J_k(i) = \text{ optimal cost of getting from } i \text{ to } t \text{ in } N-k \text{ moves,}$$

then the cost of the optimal path from i to t is $J_0(i)$
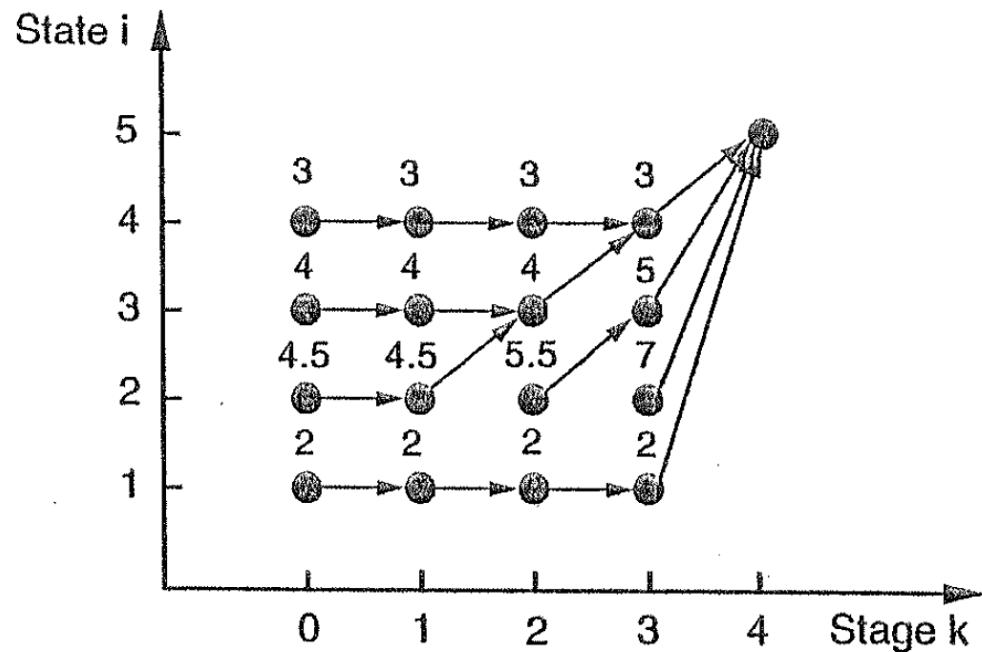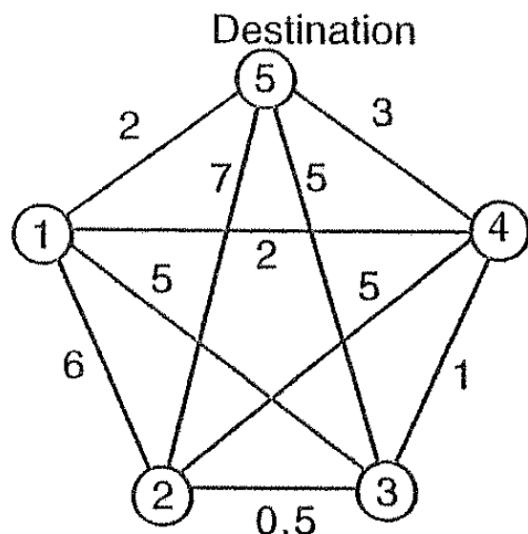
- the DP equation:

$$J_k(i) = \min_{j=1,\ldots,N} [a_{ij} + J_{k+1}(j)], \quad k = 0, 1, \ldots, N-2,$$
$$J_{N-1}(i) = a_{it}, \quad i = 1, 2, \ldots, N$$
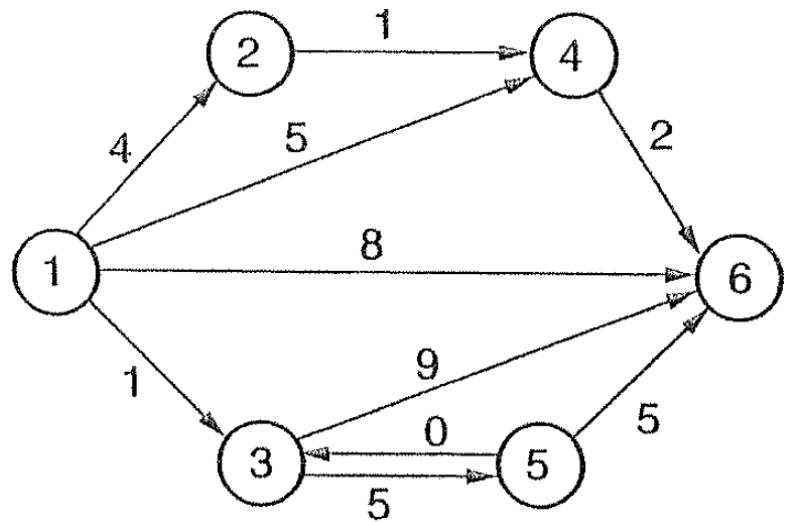
- if the optimal path obtained from the algorithm contains degenerate moves from a node to itself, this simply means that the path involves in reality less than N moves

- consider the problem below (the left figure) where the costs a_{ij} with $i \neq j$ are shown along the line segments (assume $a_{ij} = a_{ji}$)

- the right figure shows the cost-to-go $J_k(i)$ at each i and k together with the optimal paths

Example 2.2 – Shortest Path as DP

- find a shortest path from each node to node 6 by using the DP algorithm

- here: $t = 6, N = 5$

i	J_1	J_2	J_3	J_4
1				
2				
3				
4				
5				

i	μ_1	μ_2	μ_3	μ_4
1				
2				
3				
4				
5				

Hidden Markov Models and the Viterbi Algorithm

- consider a Markov chain with a finite number of states and given state transition probabilities p_{ij}

- suppose that when a transition occurs, the states corresponding to the transition are unknown (or "hidden") to us, but instead we obtain an observation that relates to the transition

- given a sequence of observations, we want to estimate in some optimal sense the sequence of corresponding transitions

- we are given the probability $r(z; i, j)$ of an observation taking value z when the state transition is from i to j and assume these observations are independent (p_{ij} and $r(z; i, j)$ also assumed independent)

- we are also given the probability π_i that the initial state takes value i

- Markov chains whose state transition are imperfectly observed are called Hidden Markov Models (HMM) or partially observable Markov chains

- we use a "most likely state" estimation criterion: given the observation sequence $Z_N = \{z_1, z_2, \ldots, z_N\}$, we adopt as our estimate the state transition sequence $\hat{X}_N = \{\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_N\}$ that maximizes over all $X_N = \{x_0, x_1, \ldots, x_N\}$ the conditional probability

$$p(X_N \mid Z_N) = \frac{p(X_N, Z_N)}{p(Z_N)}$$

- since $p(Z_N)$ is a positive constant once Z_N is known, we can maximize $p(X_N, Z_N)$ in place of $p(X_N \mid Z_N)$

- the probability $p(X_N, Z_N)$ can be written as

$$
\begin{aligned}
p(X_N, Z_N) &= p(x_0, x_1, \ldots, x_N, z_1, z_2, \ldots, z_N) \\
&= \pi_{x_0} p(x_1, \ldots, x_N, z_1, z_2, \ldots, z_N \mid x_0) \\
&= \pi_{x_0} p(x_1, z_1 \mid x_0) p(x_2, \ldots, x_N, z_2, \ldots, z_N \mid x_0, x_1, z_1) \\
&= \pi_{x_0} p_{x_0 x_1} r(z_1; x_0, x_1) p(x_2, \ldots, x_N, z_2, \ldots, z_N \mid x_0, x_1, z_1)
\end{aligned}
$$

- next steps:

$$p(x_2, \ldots, x_N, z_2, \ldots, z_N \mid x_0, x_1, z_1)$$

$$= p(x_2, z_2 \mid x_0, x_1, z_1)p(x_3, \ldots, x_N, z_3, \ldots, z_N \mid x_0, x_1, z_1, x_2, z_2)$$

$$= p_{x_1 x_2} r(z_2; x_1, x_2)p(x_3, \ldots, x_N, z_3, \ldots, z_N \mid x_0, x_1, z_1, x_2, z_2)$$

(follows from independence: $p(z_2 \mid x_0, x_1, x_2, z_1) = r(z_2; x_1, x_2)$)

- combining the two:

$$p(X_N, Z_N) = \pi_{x_0} p_{x_0 x_1} r(z_1; x_0, x_1)p_{x_1 x_2} r(z_2; x_1, x_2)$$

$$\cdot p(x_3, \ldots, x_N, z_3, \ldots, z_N \mid x_0, x_1, z_1, x_2, z_2)$$
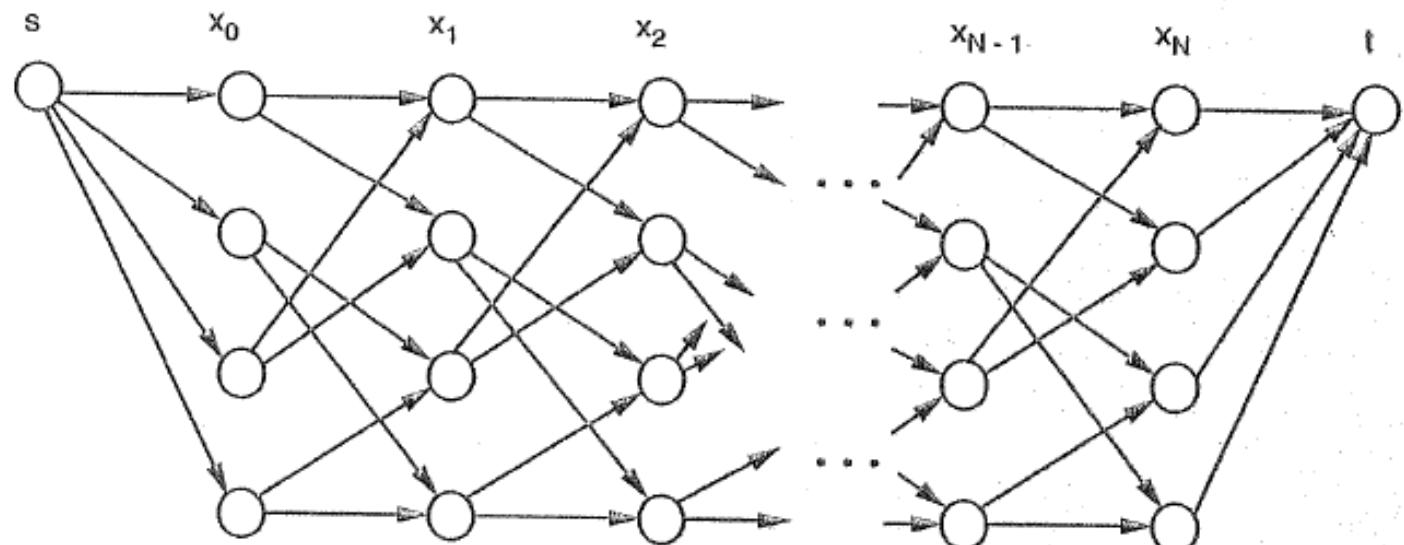
- and continuing in the same manner, we get

$$p(X_N, Z_N) = \pi_{x_0} \prod_{k=1}^{N} p_{x_{k-1} x_k} r(z_k; x_{k-1}, x_k)$$

- for maximization of the above, we can use a log transformation:

$$\ln(p(X_N, Z_N)) = \ln(\pi_{x_0}) + \sum_{k=1}^{N} \ln(p_{x_{k-1} x_k} r(z_k; x_{k-1}, x_k))$$

- the maximization of the above preceding expression can be viewed as a shortest path problem

- we construct a graph of state-time pairs, called trellis diagram, by concatenating $N+1$ copies of the state space, and by preceding and following them with dummy nodes s and t

- the nodes of the kth copy correspond to the states x_{k-1} at time $k-1$

- an arc connects a node x_{k-1} of the kth copy with a node x_k of the $(k+1)$st copy if the corresponding transition probability $p_{x_{k-1}x_k}$ is positive

- the optimization problem can be formulated as:
$$\text{minimize} \ -\ln(\pi_{x_0}) - \sum_{k=1}^{N} \ln(p_{x_{k-1}x_k} r(z_k; x_{k-1}, x_k))$$
 over all possible sequences $\{x_0, \ldots, x_N\}$
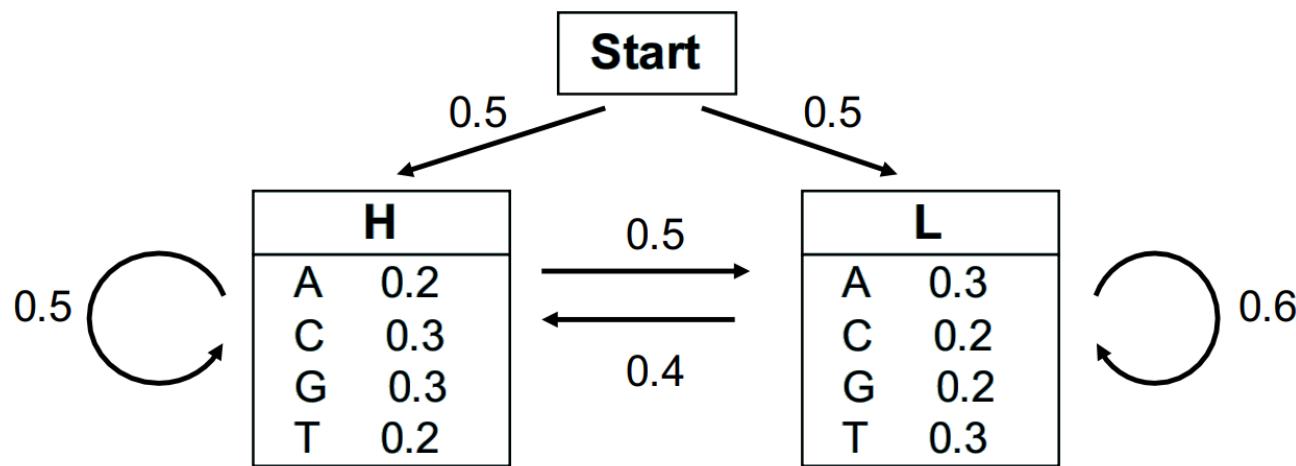
- by assigning to an arc (s, x_0) the length $-\ln(\pi_{x_0})$, to an arc (x_N, t) the length 0, and to an arc (x_{k-1}, x_k) the length $-\ln(p_{x_{k-1}x_k} r(z_k; x_{k-1}, x_k))$, we see that the above minimization problem is equivalent to the problem of finding the shortest path from s to t in the trellis diagram

- this shortest path defines the estimated state sequence $\{\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_N\}$

- in practice, the shortest path is most conveniently constructed sequentially by a forward DP: Suppose we have computed the shortest distances $D_k(x_k)$ from s to all states x_k $(D_0(x_0) = -\ln(\pi_{x_0}))$ based on the sequence Z_k, and the new observation z_{k+1} is obtained, then $D_{k+1}(x_{k+1})$ is:
$$D_{k+1}(x_{k+1}) = \min_{x_k : p_{x_k x_{k+1}} > 0} \left[D_k(x_k) - \ln(p_{x_{k-1}x_k} r(z_k; x_{k-1}, x_k)) \right],$$

- the above procedure is known as the **Viterbi algorithm** (after A. Viterbi)
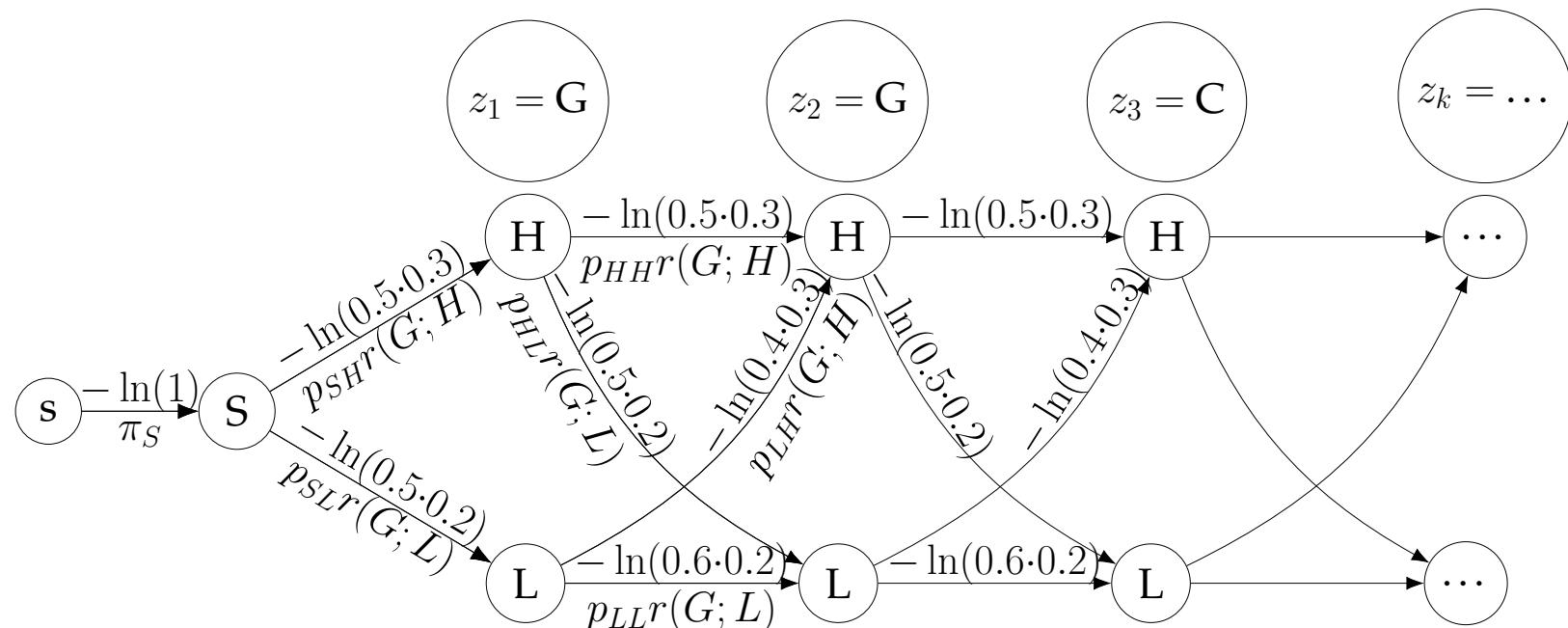
Example 2.3 – DNA Coding

- consider a simple HMM: there are two hidden states H (high CG content) and L (low CG content) (we can for example consider that state H characterizes coding DNA while L characterizes non-coding DNA)

- the model probabilities are summarized in the following graph:

- i.e., $\pi_S = 1, p_{SH} = p_{SL} = 0.5, p_{HH} = 0.5, p_{HL} = 0.5, p_{LH} = 0.4, p_{LL} = 0.6,$
  $r(A; L, H) = r(A; H, H) = 0.2, r(T; H, L) = r(T; L, L) = 0.3, \ldots$

- suppose we see the following sequence: $Z_N = \text{GGCACTGAA}$, what is the most probable sequence of the hidden states (H, L)?

CPM & PERT

- there are several methods for project evaluation:

  - Gantt charts [H. L. Gantt, cca 1910]

  - flowcharts [F. B. Gilbreth, cca 1921]

  - methods based on network analysis:

    * Critical Path Method (CPM) [M. R. Walker, J. E. Kelley, cca 1950]

    * Program Evaluation and Review Technique (PERT) [US Navy, 1958]

  - the network based methods are especially suited for the development of a work plan, control of the fulfilment of said work plan, and decision-support in situations when the work is delayed

  - the network based methods use a variant of the DP algorithm in the evaluation

Before diving into these methods, we need to be clear on the vocab:
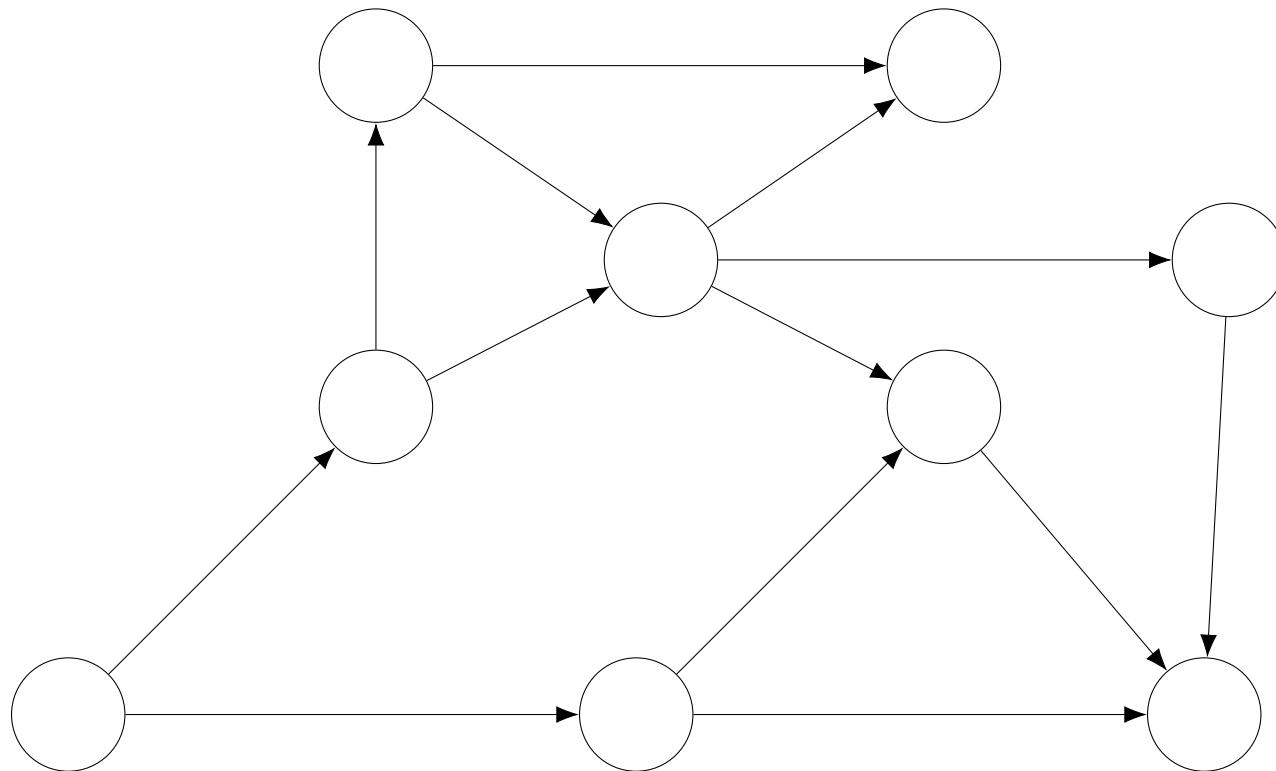
- **project** – time and space delimited set of activities, that need to be finished in order to reach a certain goal
- **network graph** – mathematical model of the project (directed acyclic simple graph with weighted edges), where individual activities are represented as edges and nodes(or vertices) represent the project state
- **undirected graph** – $G = (V, E)$, where V is a set of nodes/vertices and $E \subseteq \{\{u, v\} \mid u, v \in V\}$ is a set of two-element sets (undirected edges)
- **directed graph** – $G = (V, E)$, where V is a set of nodes and $E \subseteq V^2 = V \times V$ is a set of ordered pairs (directed edges)
- **multigraph** – with parallel edges
- **simple graph** – without parallel edges
- **walk** – a sequence of edges which joins a sequence of nodes
- **path** – a walk in which all nodes (and therefore also all edges) are distinct
- **connected graph** – there is a path between any two nodes
- **cycle** – a path that starts and finishes in the same node
- **acyclic graph** – directed graph that has no cycles
- **weighted graph** – there is a mapping $w : E \to \mathbb{R}$ for all edges

- directed acyclic graph (DAG) $\Longleftrightarrow$ there is a topological sorting/ordering (a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering)

Topological ordering algorithm

S1. Let graph $G_1 = G$ and $k = 1$.

S2. In G_k find a node that has no incoming edges and associate with it the number k. If there is no such node, the graph is not acyclic and the algorithm terminates.

S3. Get G_{k+1} from G_k by deleting the node with number k and all egdes that start at this node.

S4. If $G_{k+1} \neq \emptyset$, let $k = k + 1$ and go to S2. Else, terminate (the graph is acyclic and we get the topological ordering).


Example 2.4 – Topological Ordering
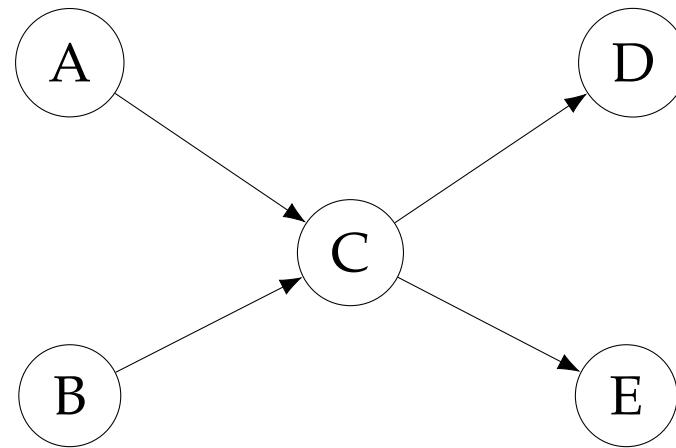
The process of network analysis:

1. Separation of the project into individual activities.

2. Estimation of the time and resources needed for the activities.

3. Finding the time dependencies between the activities.

4. **Construction of the network graph.**

5. **Time analysis (finding the critical path)**

6. Resource analysis (when having limited resources)

7. Cost analysis (finding balance between cost for project realization in a given time frame versus the possible penalties for delays)

   More in-depth analysis can be found in J. Volek, B. Linda: *Teorie grafů – aplikace v dopravě a veřejné správě*, 2012 (Univerisity of Pardubice), or in Vítečková, et al.: *Základy teorie pravděpodobnosti a teorie grafů*, 2006 (VŠB-TUO)

The interpretation of the nodes of the network graph is the following:

- conjunctive input – a node (project status) is reached when all activities (arcs) that are coming to the node are finished (this means that the outgoing activities cannot start before all incoming are finished)

- deterministic output – if a node is reached, all outgoing activities will start

- this is the interpretation used in CPM & PERT

Construction of the network graph

- in order to have the project represented by the network graph (directed acyclic simple graph with weighted edges and only one starting and one terminal node), some modifications may be needed – **fictitious** activities

- in case of multiple paths from one node to another – introduce a new node and a zero cost fictitious activity (dashed line):

- similarly in the case of multiple starting/terminal nodes

Critical Path Method

Assumptions:

- network graph (with all the conditions satisfied)

- topological ordering of the nodes

Steps:

1. Computation of the earliest attainable times for nodes.

2. Computation of the latest feasible times for nodes.

3. Computation of the time reserves.

4. Identification of the critical path and the subcritical activities.

Additional notation:
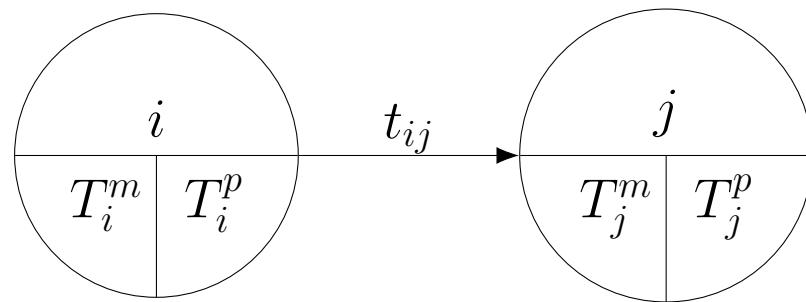
V : set of nodes, $V = 1, \ldots, n$

E : set of activities (arcs), $E \subset V \times V$

t_{ij} : duration of the activity (i, j) (arc length)

T_i^m : earliest attainable time of node i

T_i^p : latest feasible time of node i

Computation of the earliest attainable times for nodes (forward pass):

1. Set $T_1^m = 0$.

2. $T_j^m = \max_{(i,j) \in E} (t_{ij} + T_i^m), \quad j = 2, \ldots, n.$

Computation of the latest feasible times for nodes (backward pass):

1. Set $T_n^p = T_n^m$.

2. $T_i^p = \min_{(i,j) \in E}(T_j^p - t_{ij}), \quad i = n - 1, \ldots, 1.$
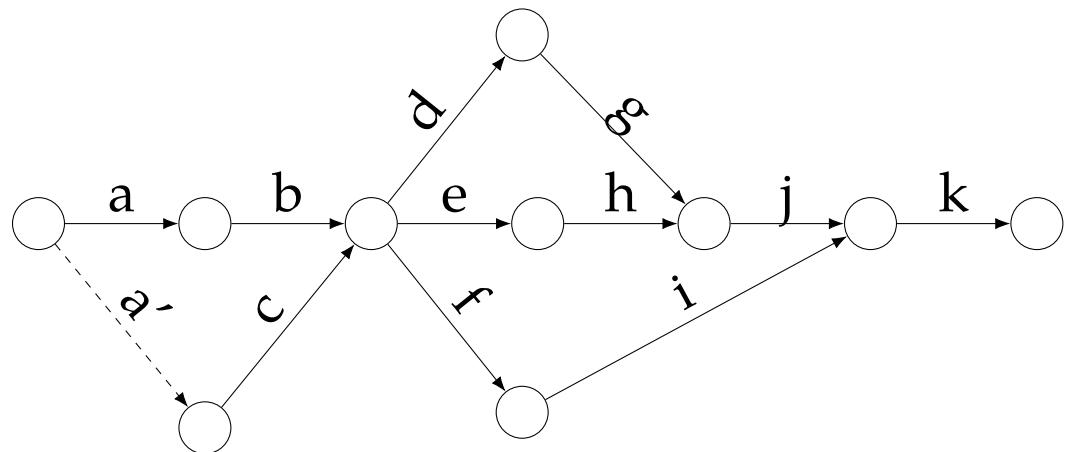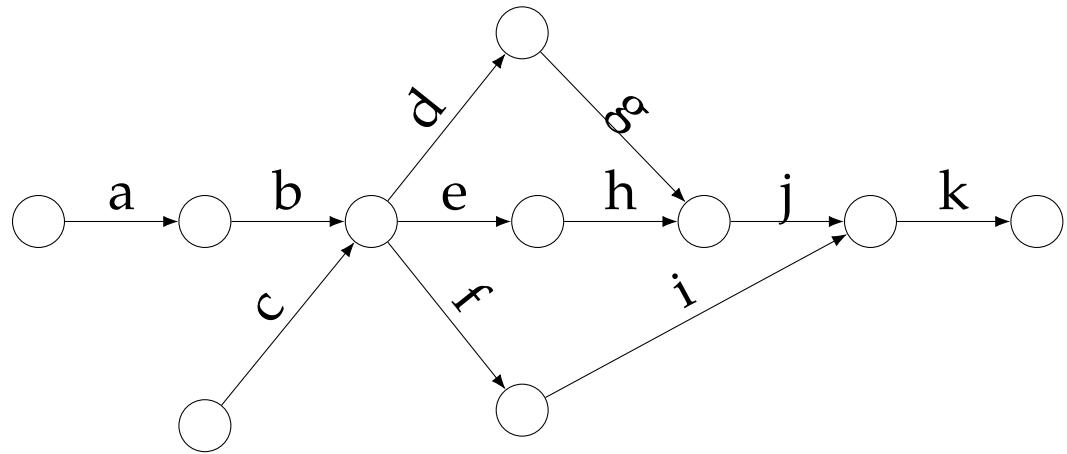
Computation of the time reserves of activity (i, j):

- total time reserve: $R_{ij}^c = T_j^p - T_i^m - t_{ij}$

- free time reserve: $R_{ij}^v = T_j^m - T_i^m - t_{ij}$

- independent time reserve: $R_{ij}^n = \max\{0, T_j^m - T_i^p - t_{ij}\}$

- dependent time reserve: $R_{ij}^z = T_j^p - T_i^p - t_{ij}$
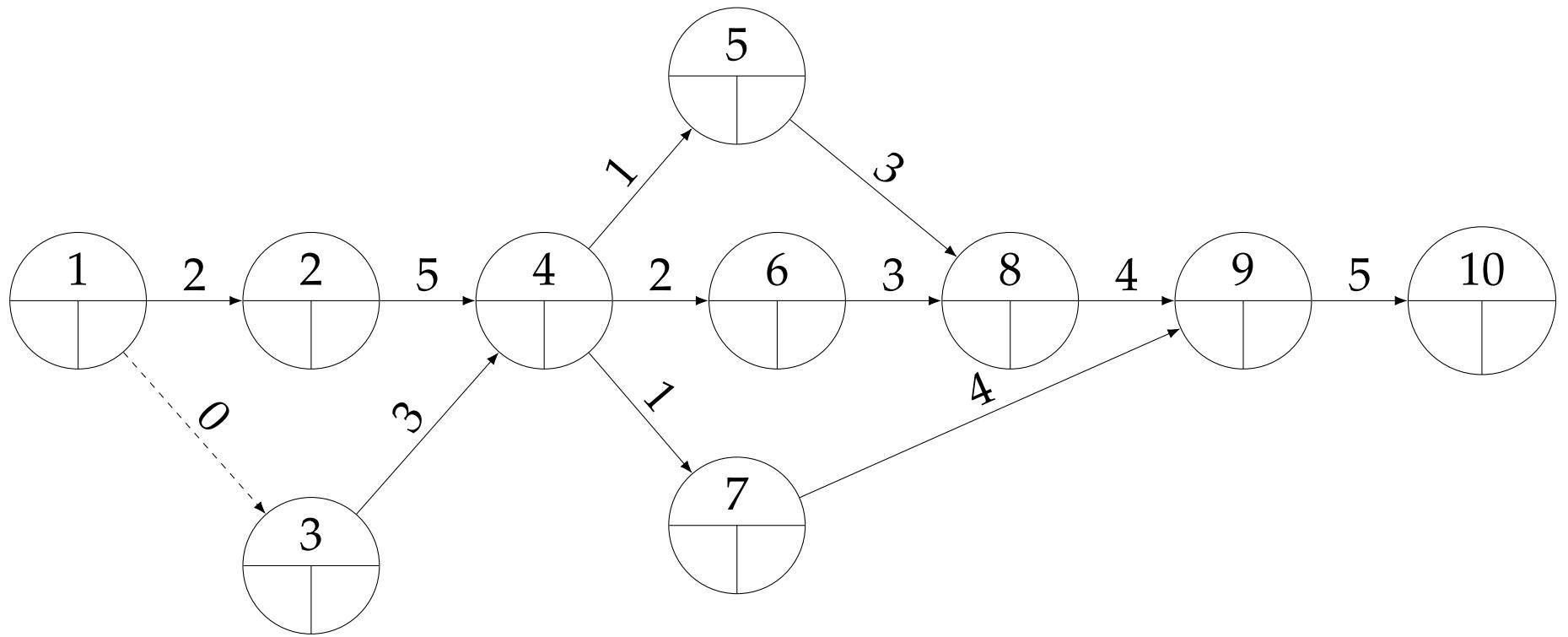
Identification of the critical path and the subcritical activities:

- **critical activity** – activity (arc) that has 0 total time reserve

- **critical path** – sequence (path) of critical activities, there always is at least one (it is the longest path in the graph from the starting node to the destination)

- **subcritical activity** – activity that has total time reserve less than some value δ

Example 2.5 – CPM

activity	id	t	cond
design	a	2	a<b
project	b	5	b<d,e,f
market an.	c	3	c<d,e,f
order1	d	1	d<g
order2	e	2	e<h
order3	f	1	f<i
deliver1	g	3	g<j
deliver2	h	3	h<j
deliver3	i	4	i<k
assemble1	j	4	j<k
assemble2	k	5	–
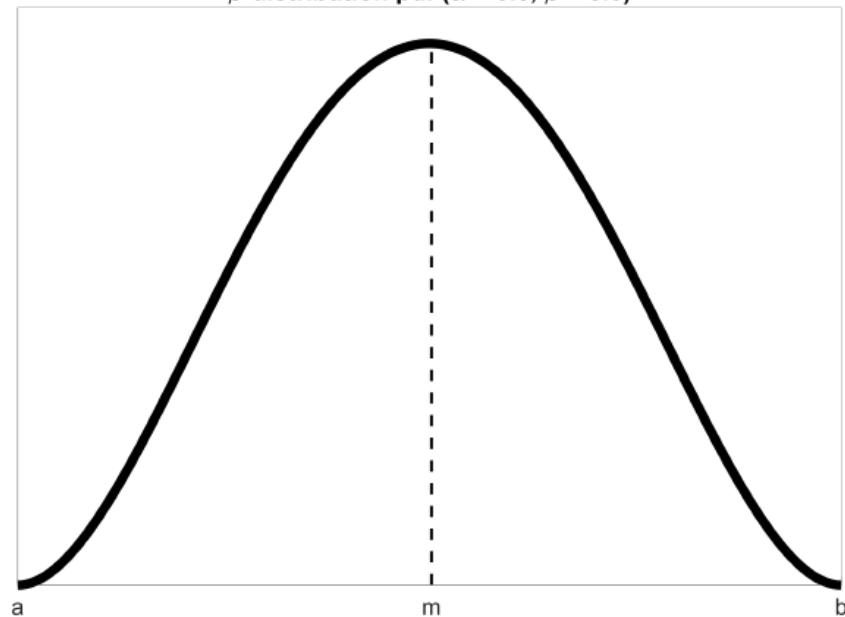
id	a	a′	b	c	d	e	f	g	h	i	j	k
(i,j)	(1,2)	(1,3)	(2,4)	(3,4)	(4,5)	(4,6)	(4,7)	(5,8)	(6,8)	(7,9)	(8,9)	(9,10)
R_{ij}^c												
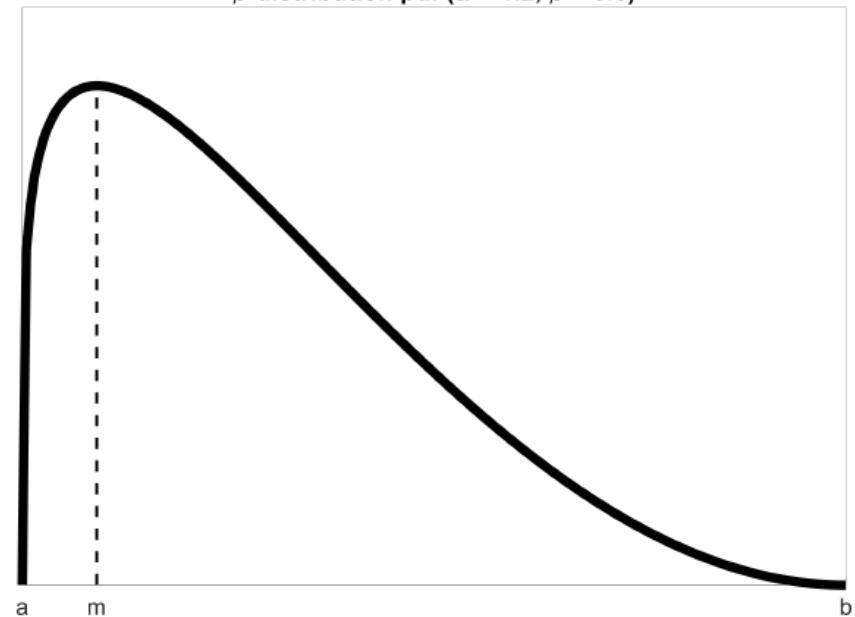
Program Evaluation and Review Technique

- one of the downsides of CPM is that it assumes a fixed (deterministic) duration for all activities

- in PERT this assumption is weakened – the duration of the activity can be a random variable, but independence is assumed

- moreover, the time duration of an activity is assumed to follow the β-**distribution** (one of the most important distributions used in Bayesian inference)

- β-distribution is two parametric, has finite support (limited max and min value), and "single" mode
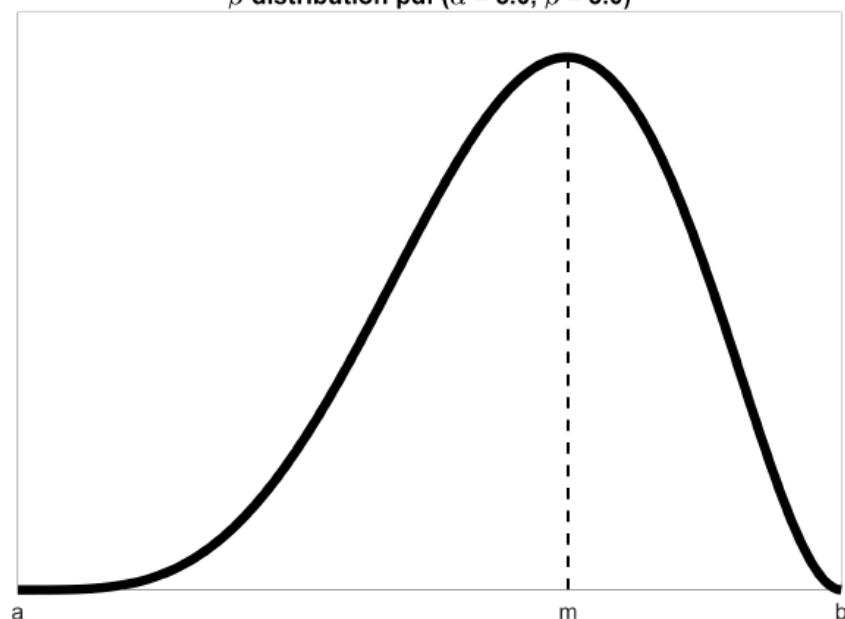
β-distribution pdf (α = 3.0, β = 3.0)
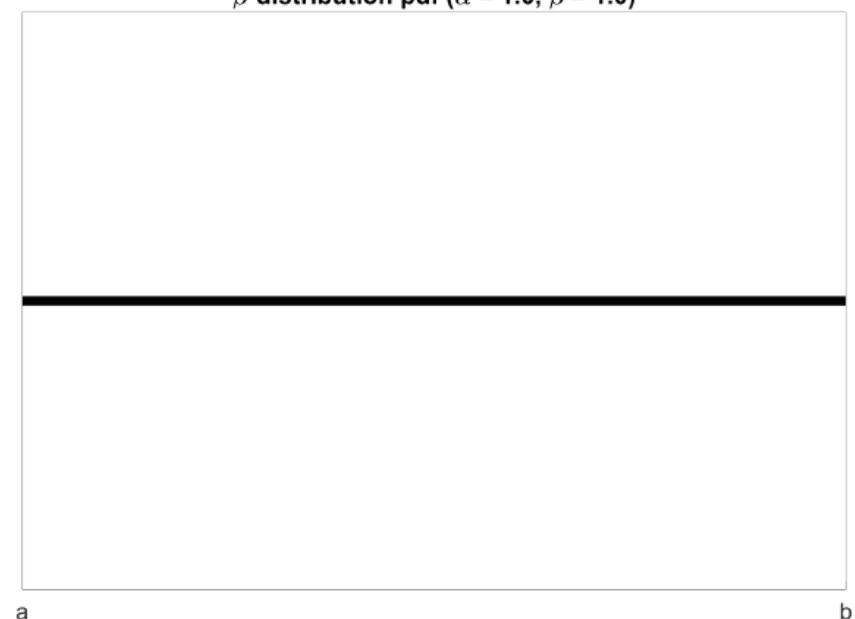
β-distribution pdf (α = 1.2, β = 3.0)

β-distribution pdf (α = 5.0, β = 3.0)

β-distribution pdf (α = 1.0, β = 1.0)

- the PERT method does not use the "natural" parameters for the β-distribution (α and β) for its identification, but instead uses the minimum value a, maximum value b and mode m, with the following interpretations

- a – optimistic estimate of the duration of the activity – "if we were to repeat the activity a hundred time, this is the best we could do"

- m – most probable estimate of the duration of the activity (self-explanatory)

- b – pessimistic estimate of the duration of the activity – "if we were to repeat the activity a hundred time, this is the worst we could do" (does not consider "vis major", like the end of the world, pandemics, etc.)

- based on these estimates of the shape of the β-distribution of the duration of the activity τ, we can compute its characteristics as

$$E[\tau] = \frac{a + 4m + b}{6}, \quad D[\tau] = \frac{(b-a)^2}{36}, \quad \sigma[\tau] = \frac{b-a}{6}$$

- similarly to CPM, the depencence of the activities is represented by a network graph

- the duration of the activities is represented by its expected value $E[\tau]$

- in the same vein, for the earliest attainable time, latest feasible time, and different time reserves, their respective expected values will be computed

- in order to obtain probabilities that the project will be completed on time or with some reserve, we need to know the distribution of these random variables (no longer just β)

- to do this, we will use the central limit theorem (more specifically, Lyapunov CLT, although there are some more involved conditions that we assume are satisfied) – the distribution of the sum of independent random variables converges to the normal distribution

- let τ_h be the duration of activity h, $m^*[v_0, v_i]$ the maximum path length (duration) from the starting node to a node i, and $m^*[v_i, v_n]$ the maximum path length from a node i to the terminal node n

- then, it is straightforward to see that

$$T_i^m = \sum_{h \in m^*[v_0, v_i]} \tau_h,$$

$$T_i^p = T_n^p - \sum_{h \in m^*[v_i, v_n]} \tau_h,$$

$$R_i = T_i^p - T_i^m,$$

which are all just sums of variables from a β-distribution

- it is quite important to note that for the CLT to "hold" (more informally: be useful), the number of activities should be rather large; then:

$$T_i^m \sim \mathcal{N}(E[T_i^m], \sqrt{D[T_i^m]}), \ T_i^p \sim \mathcal{N}(E[T_i^p], \sqrt{D[T_i^p]}), \ R_i \sim \mathcal{N}(E[R_i], \sqrt{D[R_i]})$$

- to compute the characteristics, we use their properties and the assumed independence of the individual τ_h:

$$E[T_i^m] = \sum_{h \in m^*[v_0, v_i]} E[\tau_h], \qquad D[T_i^m] = \sum_{h \in m^*[v_0, v_i]} D[\tau_h],$$

$$E[T_i^p] = T_n^p - \sum_{h \in m^*[v_i, v_n]} E[\tau_h], \qquad D[T_i^p] = \sum_{h \in m^*[v_i, v_n]} D[\tau_h],$$

$$E[R_i] = E[T_i^p] - E[T_i^m], \qquad D[R_i] = D[T_i^p] + D[T_i^m]$$

- because all of our random variables are continuous, their realized values will most certainly be different from their expected values (and, since normality is assumed, they can attain "any" value) – it is plausible that even though $E[R_i] > 0$ for some i, the real time reserve in node i can be negative, i.e. some non-critical activities may become critical and the project may be delayed

- to evaluate this risk, we can compute the probability of the negative time reserve for any node i as

$$P\{R_i < 0\} = P\left\{\frac{R_i - E[R_i]}{\sqrt{D[R_i]}} < \frac{0 - E[R_i]}{\sqrt{D[R_i]}}\right\} = \Phi\left(\frac{-E[R_i]}{\sqrt{D[R_i]}}\right),$$

  where Φ is the distribution function of the standard normal distribution $\mathcal{N}(0,1)$

- the probability of negative time reserve for nodes lying on the critical path is always 0.5

- in a similar fashion, we can check the probability of a node i being achieved before some planned time T^{v_i}, i.e. that $T_i^m < T^{v_i}$:

$$P\{T_i^m < T^{v_i}\} = \Phi\left(\frac{T^{v_i} - E[T_i^m]}{\sqrt{D[T_i^m]}}\right),$$

  where for $i = n$ we get the probability of finishing the project on time (before some T^{v_n})

- apart from controlling the probability of finishing some activities before some predetermined time, the PERT method also enables us to plan the project in such a way, that it will finish on time with a prescribed probability

- if we want to plan the project deadline T_{v_n} such that it is satisfied with probability α, we need
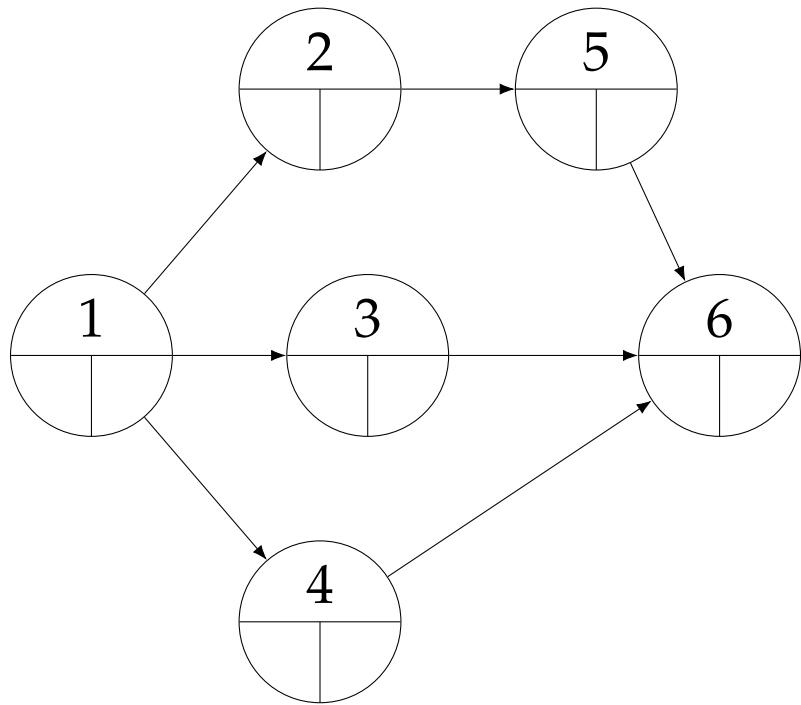
$$P\{T_n^m \leq T^{v_n}\} = \Phi\left(\frac{T^{v_n} - E[T_n^m]}{\sqrt{D[T_n^m]}}\right) = \alpha,$$

which yields

$$T^{v_n} = E[T_n^m] + x_\alpha \sqrt{D[T_n^m]},$$

where $x_\alpha = \Phi^{-1}(\alpha)$ is the α-quantile of $\mathcal{N}(0, 1)$

Example 2.6 – PERT

(i,j)	a	m	b	$E[\tau]$	$D[\tau]$
(1,2)	4	6	8		
(1,3)	1	2	3		
(1,4)	6	10	12		
(2,5)	2	3	4		
(3,6)	5	8	9		
(4,6)	3	4	5		
(5,6)	3	4	5		

$$E[\tau] = \frac{a + 4m + b}{6}$$

$$D[\tau] = \frac{(b-a)^2}{36}$$

$$P\{T_6^m < 14\} =$$

$$P\{T_6^m \leq T^{v_n}\} = 0.95 \Rightarrow T^{v_6} =$$

$$(x_{0.95} = 1.645)$$

Shortest Path Algorithms

- shortest path problems and deterministic finite-state optimal control problems are equivalent

- the computational implications are twofold:

  (a) use DP to solve general shortest path problems, but other shortest path methods may work better (have better worst-case performance); DP is preferred particularly for with an acyclic graph structure
  (b) use general shortest path methods (other than DP) for deterministic finite-state optimal control problems (sometimes preferable)

- shortest path problem with a very large number of nodes, with only one origin and one destination node – it is often true that most of the nodes are not relevant to the shortest path problem (they are unlikely candidates for the inclusion to the optimal path), but in the DP algorithm every node and arc is used in the computation

Assumptions:

- special node s called the **origin**

- special node t called the **destination** (single one)

- a node j is called a **child** of node i if there is an arc (i, j) that connects i with j

- the length of arc (i, j) is denoted by a_{ij} and we assume that all arcs have nonnegative lengths (stronger assumption than the nonnegative cycles)

- we wish to find a shortest path from origin to destination

(the upcoming theory is based on the Bertsekas' book; other sources may group/treat the methods in a bit different way)

Label Correcting Methods

- a general type of shortest path algorithms

- the main idea is to progressively discover shorter paths from the origin to every other node i, and to maintain the length of the shortest path found so far in a variable d_i called the **label of** i

- each time d_i is reduced after a discovery of a shorter path to i, the algorithm checks if the labels d_j of the children j of i can be "corrected" – reduced to $d_i + a_{ij}$

- the label d_t of the destination is maintained in a variable called $UPPER$

- the label d_s of the origin is initialized at 0 are remains at 0

- the labels of all other nodes are initialized at ∞ for all $i \neq s$

- a candidate list of nodes, called $OPEN$ – contains nodes that are currently active (candidates for further examination, possible inclusion to the shortest path)

- initially, $OPEN$ contains just the origin node s

- each node that has entered $OPEN$ at least once, except s, is assigned a "parent", which is some other node (these are not necessary for the computation of the path but for its tracing)

Label Correcting Algorithm:

S1 Remove node i from $OPEN$ and for each child j of i, do S2.

S2 If $d_i + a_{ij} < \min\{d_j, UPPER\}$, set $d_j = d_i + a_{ij}$ and set i to be the parent of j. In addition, if $j \neq t$, place j in $OPEN$ if it is not already in $OPEN$, while if $j = t$, set $UPPER$ to the new value $d_i + a_{it}$ of d_t.

S3 If $OPEN$ is empty, terminate; else go to S1.

- throughout the algorithm, d_j is either ∞ (if node j has not yet entered the $OPEN$ list), or else it is the length of some path from s to j consisting of nodes that have entered $OPEN$ at least once

- $UPPER$ is either ∞ or else it is the length of some path from s to t

- the idea of the algorithm is that when a path from s to j is discovered, which is shorter than those considered earlier ($d_i + a_{ij} < d_j$ in S2), the value of d_j is reduced, j enters $OPEN$ and i is set to be the parent of j – however, this makes sense only when $d_j < UPPER$

- when the algorithm terminates a shortest path can be obtained by tracing backwards the parent nodes starting from t and going towards s

- **if there exists a path at least one path from s to t, the label correcting algorithm terminates with $UPPER$ equal to the shortest distance from s to t, otherwise the algorithm terminates with $UPPER = \infty$** [Proof in Bertsekas' book, Sec. 2.3]

- an important property of the algorithm is that nodes j for which $d_i + a_{ij} \geq$ $UPPER$ in S2 will not enter $OPEN$ in the current iteration, and may possibly not enter in any subsequent iteration

- the number of nodes that enter $OPEN$ may be much smaller than the total number of nodes

- furthermore, if a good lower bound to the shortest distance from s to t (or the shortest distance itself) is known, the computation can be terminated once $UPPER$ reaches that bound within an acceptable tolerance

- the specific label correcting methods differ in the way they select a node to be removed from $OPEN$ at each iteration
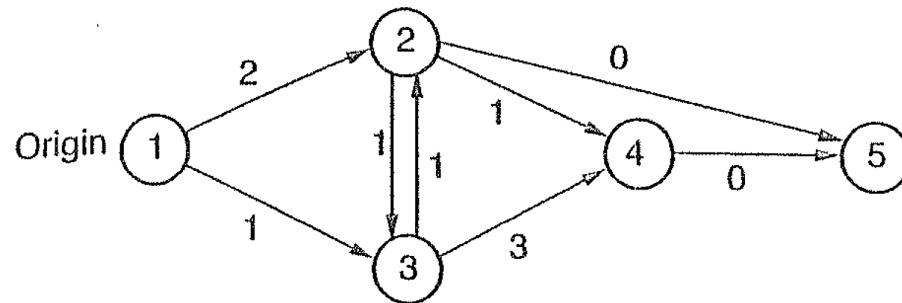
Specific label correcting methods:

(a) **Breath-first search** (also known as Bellman-Ford method) adopts a first-in/first-out policy – node is always removed from the top of $OPEN$ and each node entering $OPEN$ is placed at the bottom

(b) **Depth-first search** adopts a last-in/first-out policy – node is always removed from the top of $OPEN$ and each node entering $OPEN$ is placed at the top

(c) **Best-first search** (also known as Dijkstra's method) – with each iteration removes from $OPEN$ a node with minimum value of label, i.e. a node i with

$$d_i = \min_{j \in OPEN} d_j$$

(d) **D'Esopo-Pape method** – node is always removed from the top of $OPEN$ and, but inserts a node at the top of $OPEN$ if it has already been in $OPEN$ earlier, and inserts the node at the bottom of $OPEN$ otherwise

Example 2.7 – Bellman-Ford

iter	node exiting OPEN	OPEN at the end of iter	d_1	d_2	d_3	d_4	UPPER
0	–	1	0	∞	∞	∞	∞
1							
2							
3							
4							

Label correcting variations – A^* algorithm

- the generic label correcting algorithm need not be started with the initial conditions $d_s = 0$ and $d_i = \infty$ for $i \neq s$ in order to work correctly – it can be shown that one can use any set of initial labels d_i such that for each node i, d_i is either ∞ or else it is the length of some path from s to i, the scalar $UPPER$ may be taken to be equal d_t and the initial $OPEN$ list may be taken to be the set $\{i \,|\, d_i < \infty\}$

- this kind of initialization is very useful if, by using a heuristic or a known solution of a similar shortest path problem, we can construct a "good" path $P = (s, i_1, \ldots, i_k, t)$ from s to t. Then we can initialize the algorithm with

$$
d_i =
\begin{cases}
\text{length of portion of path } P \text{ from } s \text{ to } i, & \text{if } i \in P \\
\infty, & \text{otherwise}
\end{cases}
$$

with $UPPER = d_t$ and $OPEN = \{s, i_1, \ldots, i_k\}$

- if P is a near-optimal path (and $UPPER$ is near its final value), the test for admissibility to $OPEN$ will be relatively tight from the start of the algorithm – many unnecessary entrances to $OPEN$ may be avoided

- another possibility – **the A^* algorithm** – is to strengthen the test $d_i + a_{ij} < UPPER$ that the node j must pass before it is place to $OPEN$

  – this can be done if a **positive underestimate** h_j of the shortest distance of node j to the destination is available

  – the computation may then be substantially sped up by placing node j in $OPEN$ only when

  $$d_i + a_{ij} + h_j < UPPER \qquad \text{[instead of } d_i + a_{ij} < UPPER\text{]}$$

  – in this way, fewer nodes will potentially be placed in $OPEN$ before termination
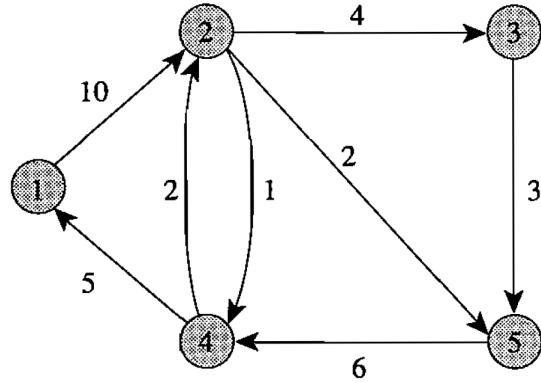
Floyd-Warshall Algorithm

- for finding a shortest $i - j$ path for each part of nodes (i, j) in the graph

- arc cost may be negative, can even detect negative cycles

- labels are for each pair of nodes, i.e. d_{ij}

- initialization: $d_{ii} = 0$, $d_{ij} = a_{ij}$ (if there is no arc from i to j, $a_{ij} = \infty$)

- the labels d_{ij} are systematically (for $k = 1, \ldots, n$, where n is the number of nodes) updated by checking the conditions

$$d_{ij} > d_{ik} + d_{kj} \text{ for some } k,$$

and if this condition is satisfied, then d_{ij} is replaced by $d_{ik} + d_{kj}$

Example 2.8 – Floyd-Warshall

d_{ij}	1	2	3	4	5
1	0	10	∞	∞	∞
2	∞	0	4	1	2
3	∞	∞	0	∞	3
4	5	2	∞	0	∞
5	∞	∞	∞	6	0

d_{ij}	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

d_{ij}	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

d_{ij}	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

d_{ij}	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

d_{ij}	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

Implementation issues (Ahuja et al.: Network Flows):

Algorithm	Running time	Features
Original implementation	$O(n^2)$	1. Selects a node with the minimum temporary distance label, designating it as permanent, and examines arcs incident to it to modify other distance labels. 2. Very easy to implement. 3. Achieves the best available running time for dense networks.
Dial's implementation	$O(m + nC)$	1. Stores the temporary labeled nodes in a sorted order in unit length buckets and identifies the minimum temporary distance label by sequentially examining the buckets. 2. Easy to implement and has excellent empirical behavior. 3. The algorithm's running time is pseudopolynomial and hence is theoretically unattractive.
d-Heap implementation	$O(m \log_d n)$, where $d = m/n$	1. Uses the d-heap data structure to maintain temporary labeled nodes. 2. Linear running time whenever $m = \Omega(n^{1+\epsilon})$ for any positive $\epsilon > 0$.
Fibonacci heap implementation	$O(m + n \log n)$	1. Uses the Fibonacci heap data structure to maintain temporary labeled nodes. 2. Achieves the best available strongly polynomial running time for solving shortest paths problems. 3. Intricate and difficult to implement.
Radix heap implementation	$O(m + n \log(nC))$	1. Uses a radix heap to implement Dijkstra's algorithm. 2. Improves Dial's algorithm by storing temporarily labeled nodes in buckets with varied widths. 3. Achieves an excellent running time for problems that satisfy the similarity assumption.

Figure 4.14   Summary of different implementations of Dijkstra's algorithm.

Algorithm	Running Time	Features
Generic label-correcting algorithm	$O(\min\{n^2mC, m2^n\})$	1. Selects arcs violating their optimality conditions and updates distance labels. 2. Requires $O(m)$ time to identify an arc violating its optimality condition. 3. Very general: most shortest path algorithms can be viewed as special cases of this algorithm. 4. The running time is pseudopolynomial and so is unattractive.
Modified label-correcting algorithm	$O(\min\{nmC, m2^n\})$	1. An improved implementation of the generic label-correcting algorithm. 2. The algorithm maintains a set, LIST, of nodes: whenever a distance label $d(j)$ changes, we add node j to LIST. The algorithm removes a node i from LIST and examines arcs in $A(i)$ to update distance labels. 3. Very flexible since we can maintain LIST in a variety of ways. 4. The running time is still unattractive.
FIFO implementation	$O(nm)$	1. A specific implementation of the modified label-correcting algorithm. 2. Maintains the set LIST as a queue and hence examines nodes in LIST in first-in, first-out order. 3. Achieves the best strongly polynomial running time for solving the shortest path problem with arbitrary arc lengths. 4. Quite efficient in practice. 5. In $O(nm)$ time, can also identify the presence of negative cycles.
Dequeue implementation	$O(\min\{nmC, m2^n\})$	1. Another specific implementation of the modified label-correcting algorithm. 2. Maintains the set LIST as a dequeue. Adds a node to the front of dequeue if the algorithm has previously updated its distance label, and to the rear otherwise. 3. Very efficient in practice (possibly, linear time). 4. The worst-case running time is unattractive.

Figure 5.8   Summary of label-correcting algorithms.

Branch-and-Bound

- consider a problem of minimizing a cost function $f(x)$ over a **finite** set of possible solutions X

- in particular, problems with very large number of feasible solutions, so an enumeration and comparison of these solutions is impractical

- the idea of the branch-and-bound method is to avoid a complete enumeration by discarding solutions that, based on certain tests, have no chance of being optimal (in this sense is similar to the label correcting methods)

- the main idea is to partition the feasible set into smaller subsets, and then use certain bounds on the attainable cost withing some of the subsets to eliminate from further consideration other subsets

Bounding Principle: Given a problem of minimizing $f(x)$ over $x \in X$, and two subsets $Y_1 \subset X$ and $Y_2 \subset X$, suppose that we have bounds

$$\underline{f}_1 \le \min_{x \in Y_1} f(x), \qquad \overline{f}_2 \ge \min_{x \in Y_2} f(x).$$

Then if $\overline{f}_2 \le \underline{f}_1$, the solutions in Y_1 may be discarded since their cost cannot be smaller than the cost of the best solution Y_2.

- the branch-and-bound method calculates suitable upper and lower bounds, and uses the bounding principle to eliminate from consideration substantial portions of the feasible set

- to describe the method, we use an acyclic graph with nodes that correspond on a one-to-one basis with a collection $\mathcal{X}$ of subsets of the feasible set X

We require the following:
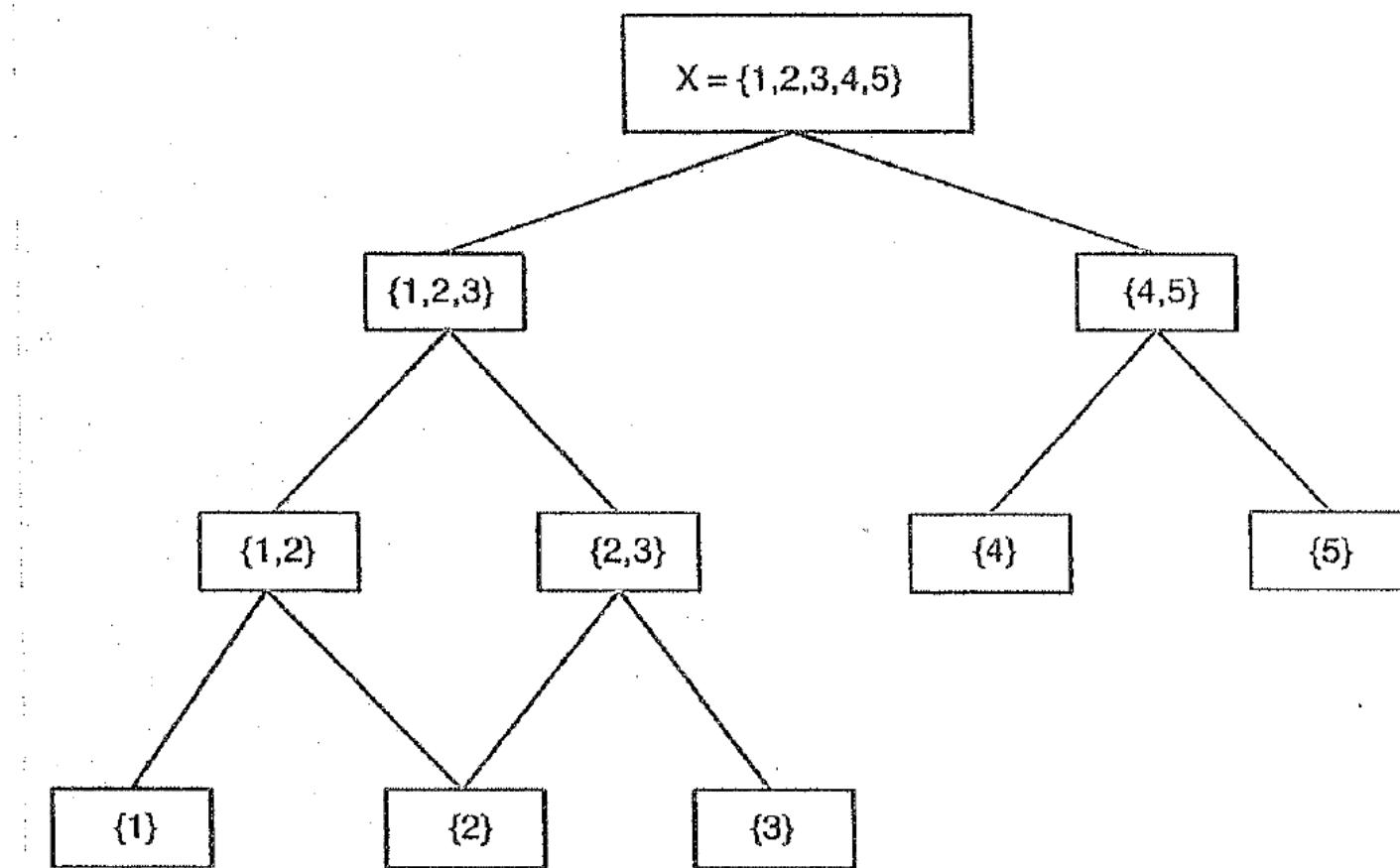
1. $X \in \mathcal{X}$ (i.e. the set of all solutions is a node).

2. For each solution x, we have $\{x\} \in \mathcal{X}$ (each solution viewed as a singleton set is a node).

3. Each set $Y \in \mathcal{X}$ that contains more than one solution $x \in X$ is partitioned into sets $Y_1, \ldots, Y_n \in \mathcal{X}$ such that $Y_i \neq Y$ for all i:

$$\bigcup_{i=1}^{n} Y_i = Y.$$

   The set Y is called the **parent** of $Y_1, \ldots, Y_n$, and the sets $Y_1, \ldots, Y_n$ are called **children** if Y.

4. Each set in $\mathcal{X}$ other than X has at least one parent.

- the collection of sets $\mathcal{X}$ defines an acyclic graph with root node the set of all feasible solutions X and terminal nodes the singleton solutions $\{x\}, x \in X$; arc of the graph are those that connect parents Y to their children Y_i

- suppose that for every nonterminal node Y there is an algorithm that calculates upper and lower bounds $\overline{f}_Y$ and $\underline{f}_Y$ for the minimum cost over Y:

$$\underline{f}_Y \leq \min_{x \in Y} f(x) \leq \overline{f}_Y$$

and assume that the upper and lower bounds are exact for each singleton solution node $\{x\}$:

$$\underline{f}_{\{x\}} = f(x) = \overline{f}_{\{x\}}, \quad \text{for all } x \in X.$$

- define the length of an arc involving a parent Y and a child Y_i to be the lower bound difference

$$\underline{f}_{Y_i} - \underline{f}_Y$$

then the length of any path from the origin node X to any node Y is $\underline{f}_Y - \underline{f}_X$ [wrong in the book]

- since $\underline{f}_{\{x\}} = f(x)$ for all feasible solutions $x \in X$, it is clear that minimizing $f(x)$ over X is equivalent to finding a shortest path from the origin to one of the singleton nodes $\{x\}$

- consider a variation of the label correcting method, where in addition we use our knowledge of the upper bounds $\overline{f}_Y$ to reduce the value of $UPPER$; initially, $OPEN = \{X\}$ and $UPPER = \overline{f}_X$

Branch-and-Bound Algorithm

S1: Remove a node Y from $OPEN$. For each child Y_j of Y, do the following: If $\underline{f}_{Y_j} < UPPER$, then place Y_j in $OPEN$. If in addition $\overline{f}_{Y_j} < UPPER$, then set $UPPER = \overline{f}_{Y_j}$, and if Y_j consists of a single solution, mark the solution as being the best solution found so far (incumbent).

S2: (Termination Test) If $OPEN$ is nonempty, go to S1. Otherwise, terminate; the best solution found so far is optimal.

- alternatively, is S2, one can set a tolerance $\varepsilon > 0$, and terminate once the difference between $UPPER$ and the minimum lower bound $\underline{f}_Y$ over all sets Y in the $OPEN$ list differs by less than ε

- important note: **it is neither practical nor necessary to generate a priori the acyclic graph of the branch-and-bound method** [one should adaptively decide, on the order and manner in which the parent sets are partitioned into children sets based on the progress of the algorithm]

- in order to effectively apply the branch-and-bound algorithm, it is important to have good algorithms for generating upper and lower bounds at each node – when these are as sharp as possible, then fewer nodes are admitted to $OPEN$

- typically, continuous optimization problems (linear or network optimization) are used to obtain lower bounds, while various heuristics are used to construct corresponding feasible solutions whose costs can be used as upper bounds

Example 2.9 – MILP

- branch-and-bound method is one of the go-to methods for mixed-integer linear programming problems, with many extensions (brand-and-cut, branch-and-price, etc.)

- getting lower bound: relaxing the integrality requirements and solving a "simple" LP

- getting upper bound: any solution that satisfies integrality constraints can serve as an upper bound (the smaller the better)

- consider the following problem:

$$\text{minimize } 7x_1 + 12x_2 + 5x_3 + 14x_4$$
$$\text{s.t. } 300x_1 + 600x_2 + 500x_3 + 1600x_4 \geq 700$$
$$x_i \in \{0, 1, 2\}, i = 1, \ldots, 4$$

Example 2.10 – Job Scheduling

- sometimes, the derivation of the lower bounds is not that straightforward and coming up with (a decent) lower bound might be quite difficult

- consider the following problem (flow-shop):
  We are given n jobs $i = 1, \ldots, n$. Each job has two operations to be performed on one of two machines. Job i requires a processing time p_{ij} on machine j ($j = 1, 2$) and each job must complete processing on machine 1 before starting on machine 2. Let C_i be the time at which job i finishes on machine 2. We have to find a schedule which minimizes the sum of finishing times $\sum_{i=1}^{n} C_i$.

- it can be shown that for the problem above, there exists an optimal schedule in which both machines process the jobs in the same order (proof in Brucker: Scheduling Algorithms), which means that an optimal schedule may be represented by a job permutation

- a natural way to branch is to choose the first job to be scheduled at the first level of the branching tree, the second job at the next level, and so on

- suppose we are at a node at which the jobs in the set $M \subseteq \{1, \ldots, n\}$ have been scheduled, where $|M| = r$. Let $i_k, k = 1, \ldots, n$, be the index of the k-th job under any schedule which is a descendant of the node under consideration

- the cost of this schedule, which we wish to bound, is

$$S = \sum_{i \in M} C_i + \sum_{i \notin M} C_i$$

where for the second sum we will derive two possible lower bounds

(1) If every job $i \notin M$ could start its processing on machine 2 immediately after completing its processing on machine 1, the second sum would become

$$S_1 = \sum_{k=r+1}^{n} [\sum_{i \in M} p_{i1} + (n - k + 1)p_{i_k 1} + p_{i_k 2}]$$

and, if that is not possible, we have

$$\sum_{i \notin M} C_i \geq S_1.$$

The bound S_1 depends on the way the jobs not in M are scheduled. This dependence can be eliminated by noting that S_1 is minimized by scheduling jobs $i \notin M$ in an order of nondecreasing p_{i1}-values. Call the resulting minimum value S_1^*.

(2) $\max\{C_{i_r}, \sum_{i\in M} p_{i1} + \min_{i\notin M} p_{i1}\}$ is a lower bound on the start of the first job $i \notin M$ on machine 2. Thus, the second sum in would be bounded by

$$S_2 = \sum_{k=r+1}^{n} [\max\{C_{i_r}, \sum_{i\in M} p_{i1} + \min_{i\notin M} p_{i1}\} + (n - k + 1)p_{i_k 2}].$$

Again, S_2 is minimized by scheduling jobs $i \notin M$ in an order of nondecreasing p_{i2}-values. Call the resulting minimum value S_2^*.

- combining the two lower bounds we get

$$\sum_{i\in M} C_i + \max\{S_1^*, S_2^*\}$$

which is an easily computed lower bound

- as an example, consider the following values of $P = [p_{ij}]$ for $n = 5$:

$$P = \begin{bmatrix} 3 & 1 & 5 & 2 & 4 \\ 1 & 4 & 1 & 2 & 3 \end{bmatrix}^T$$

Constrained and Multiobjective Problems

- in some shortest path contexts, there may be constraints on the resources required to travel the optimal path (time, fuel, etc)

- we represent path constraints in the generic form

$$\sum_{(i,j)\in P} c_{ij}^m \leq b^m, \quad m = 1, \ldots, M,$$

  where c_{ij}^m is the amount of the mth resource required to traverse arc (i,j) and b^m is the total amount of mth resource available

- the problem is now to find a path P that starts at s, ends at t, satisfies the constraints, and minimizes

$$\sum_{(i,j)\in P} a_{ij}$$

- this is called the **constrained shortest path problem**

- a closely related problem is the **constraint feasibility problem**, where we simply want to find a path P that satisfied the above mentioned constraints

- in particular, the constraint feasibility problem is a special case of the constrained shortest path problem where $a_{ij} = 0$ for all arcs (i, j)

- conversely, the constrained shortest path problem is equivalent to the constraint feasibility problem involving the above mentioned constraints **and** the additional constraint

$$\sum_{(i,j)\in P} a_{ij} \leq L^*,$$

where L^* is the optimal path length (which we generally do not know)

- another related problem is the **multiobjective shortest path problem**, where we want to find a path P that simultaneously makes all the lengths

$$\sum_{(i,j) \in P} c_{ij}^m, \quad m = 1, \dots, M,$$

"small" in the following sense

- for any set $S \subset \mathbb{R}^M$, let us call a vector $x = (x_1, \dots, x_M) \in S$ **noninferior** if x is not dominated by any vector $y = (y_1, \dots, y_M) \in S$, in the sense that

$$y_m \leq x_m, \quad m = 1, \dots, M,$$
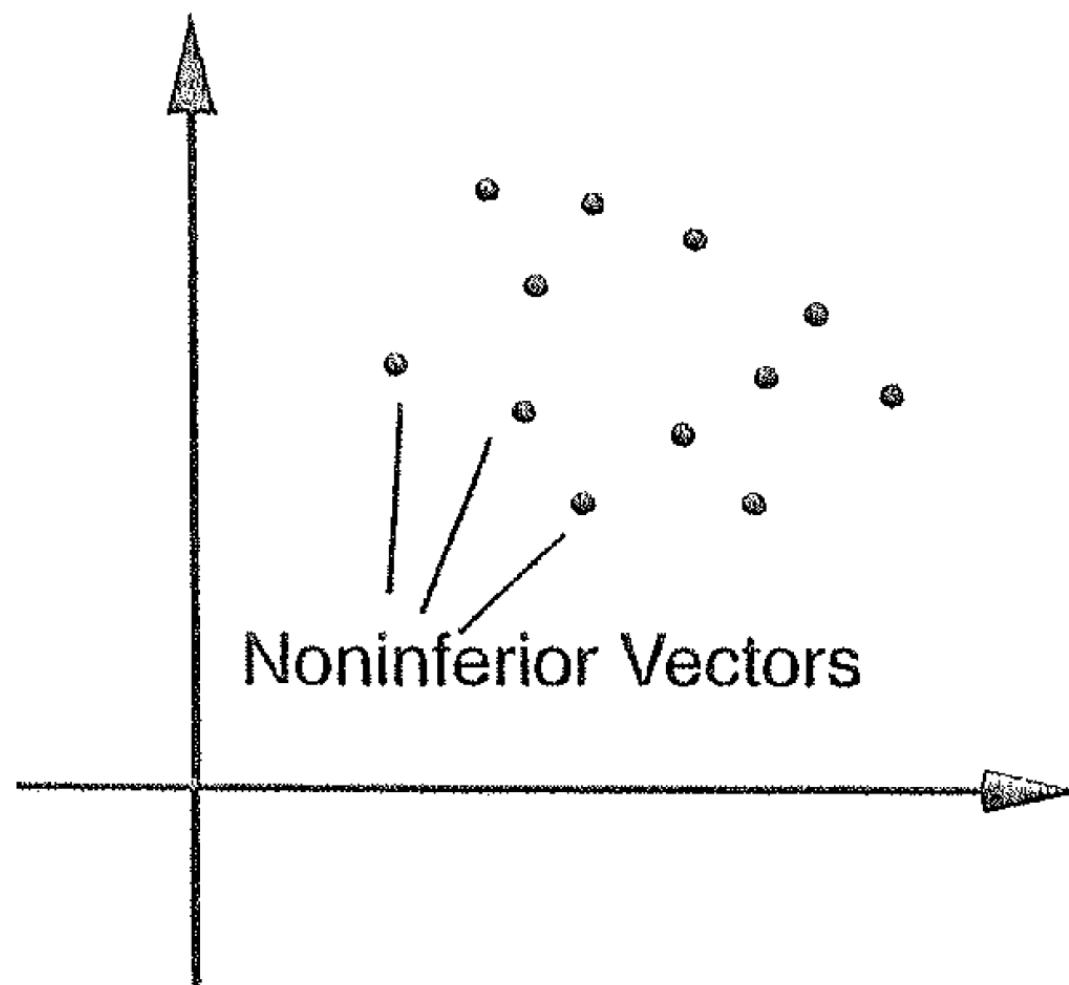
with a strict inequality for at least one m

- more generally, given a problem with multiple cost functions $f_1(x), \dots, f_M(x)$ and a constraint set X, we say that x is a **noninferior** solution if the vector of costs of x, i.e. $(f_1(x), \dots, f_M(x))$, is a noninferior vector of the set of attainable costs

$$\{(f_1(y), \dots, f_M(y)) \mid y \in X\}$$

- given a finite set of solutions, there is at least one noninferior solution

- simple algorithm for finding noninferior solutions: sequentially test all solutions and discard those that are dominated by some not yet discarded solutions, until no more solutions can be discarded

Noninferior Vectors

- the multiobjective shortest path problem is to find a noninferior path P, i.e., one for which there is no other path P' satisfying

$$\sum_{(i,j)\in P'} c_{ij}^m \leq \sum_{(i,j)\in P} c_{ij}^m, \quad m = 1, \ldots, M,$$

and with strict inequality for at least one m

- the constrained shortest path problem can be solved by casting it as a multiobjective shortest path problem, where the multiple objectives correspond to the cost and the constraints – given the set of all noninferior solutions, one obtains an optimal solution to the shortest path problem by (if a feasible solution exists), by selecting a path from this set that satisfies the constraints and minimizes the cost

- because of this connection between the problems, they fundamentally share the same mathematical structure, and can be addressed with similar methodology

Multiobjective DP Problems

- we have shown that shortest path problems and deterministic finite-state DP problems are equivalent

- it should not be surprising that the methodology for multiobjective and/or constrained shortest path problems is shared by multiobjective and/or constrained deterministic finite-state DP problems

- a multiobjective version of such problem involves a single controlled deterministic finite-state system

$$x_{k+1} = f_k(x_k, u_k),$$

and multiple cost functions of the form

$$g_N^m(x_N) + \sum_{k=0}^{N-1} g_k^m(x_k, u_k), \quad m = 1, \ldots, M.$$

- the modified DP algorithm for this problem proceeds backwards in time, and calculates for each stage k and state x_k, the set of noninferior control sequences for the tail (multiobjective) subproblem that starts at state x_k

- the algorithm is based on a fairly evident extension of the principle of optimality:
  If $\{u_k, u_{k+1}, \ldots, u_{N-1}\}$ is a noninferior control sequence for the tail subproblem that starts at x_k, then $\{u_{k+1}, \ldots, u_{N-1}\}$ is a noninferior control sequence for the tail subproblem that starts at $f_k(x_k, u_k)$.

- more specifically, let $\mathcal{F}_k(x_k)$ be the set of all M-tuples of cost-to-go

$$
\left(g_N^1(x_N) + \sum_{i=k}^{N-1} g_i^1(x_i, u_i), \ldots, g_N^M(x_N) + \sum_{i=k}^{N-1} g_i^M(x_i, u_i) \right),
$$

which correspond to control sequences $\{u_k, \ldots, u_{N-1}\}$ that start at x_k and are noninferior (note that the set $\mathcal{F}_k(x_k)$ is finite)

- the sets $\mathcal{F}_k(x_k)$ are generated by an algorithm that start at time N with $\mathcal{F}_N(x_N)$ consisting of just the vector of terminal costs,

$$\mathcal{F}_N(x_N) = \left\{ (g_N^1(x_N), \ldots, g_N^M(x_N)) \right\},$$

and proceeds backwards according to the following process: given the set $\mathcal{F}_{k+1}(x_{k+1})$ for all states x_{k+1}, the algorithm generates for each state x_k the set of vectors

$$(g_k^1(x_k, u_k) + c^1, \ldots, g_k^M(x_k, u_k) + c^M)$$

such that
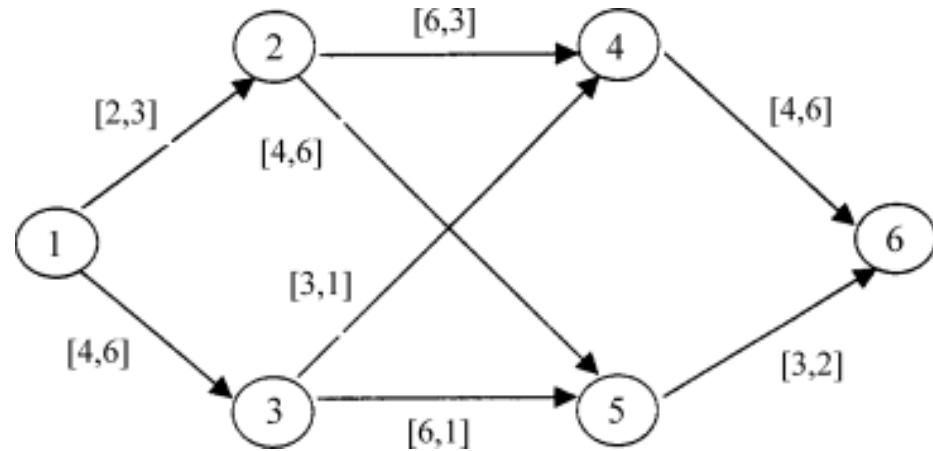
$$(c^1, \ldots, c^M) \in \mathcal{F}_{k+1}(f_k(x_k, u_k)), \quad u_k \in U_k(x_k);$$

then obtains $F_k(x_k)$ by extracting the noninferior subset, i.e., by discarding from this set the vectors that are dominated by other vectors

- after N steps, this algorithm yields $\mathcal{F}_0(x_0)$, the set of all noninferior M-tuples of cost-to-go starting at x_0

Example 2.11 – Multiobjective SP

- two objectives: path length and time

i	...	J_3	J_4	J_5
1				
2				
3				
4				
5				

i	...	μ_3	μ_4	μ_5
1				
2				
3				
4				
5				

Constrained DP Problems

- again, the same controlled system

$$x_{k+1} = f_k(x_k, u_k),$$

where we want to minimize the cost function

$$g_N^1(x_N) + \sum_{k=0}^{N-1} g_k^1(x_k, u_k)$$

subject to constraints

$$g_N^m(x_N) + \sum_{k=0}^{N-1} g_k^m(x_k, u_k) \leq b^m, \quad m = 2, \dots, M.$$

- we can solve this problem by finding the set of noninferior solutions of the multiobjective DP by extracting from this set the subset of solutions that satisfy the constraints, and by selecting from this subset the solution that minimizes the cost

- however, we can enhance this algorithm by discarding at the earliest opportunity control sequences that cannot be part of the feasible control sequence (similar to the rationale behind the A^* algorithm)

- for $m = 2, \ldots, M$ let $\tilde{J}_k^m(x_k)$ be the cost-to-arrive to x_k from the given initial state x_0 with cost per stage equal to $g_i^m(x_i, u_i)$, i.e., the minimal value of

$$\sum_{i=0}^{k-1} g_i^m(x_i, u_i), \quad m = 2, \ldots, M, \quad [\text{book error}]$$

subject to the constraint that the state at the kth stage is x_k, and can be computed by the forward DP algorithm

- consider now a DP-like algorithm that generates for each state and stage, a subset of M-tuples – it starts at N with the set $\mathcal{F}_N(x_N)$ that consists of just the vector of terminal costs

$$\mathcal{F}_N(x_N) = \{(g_N^1(x_N), \ldots, g_N^M(x_N))\},$$

132

- it then proceeds backwards as follows: given the set $\mathcal{F}_{k+1}(x_{k+1})$ for each state x_{k+1}, it generates for each state x_k the set of M-tuples

$$(g_k^1(x_k, u_k) + c^1, \ldots, g_k^M(x_k, u_k) + c^M)$$

such that

$$(c^1, \ldots, c^M) \in \mathcal{F}_{k+1}(f_k(x_k, u_k)), \quad u_k \in U_k(x_k),$$

and
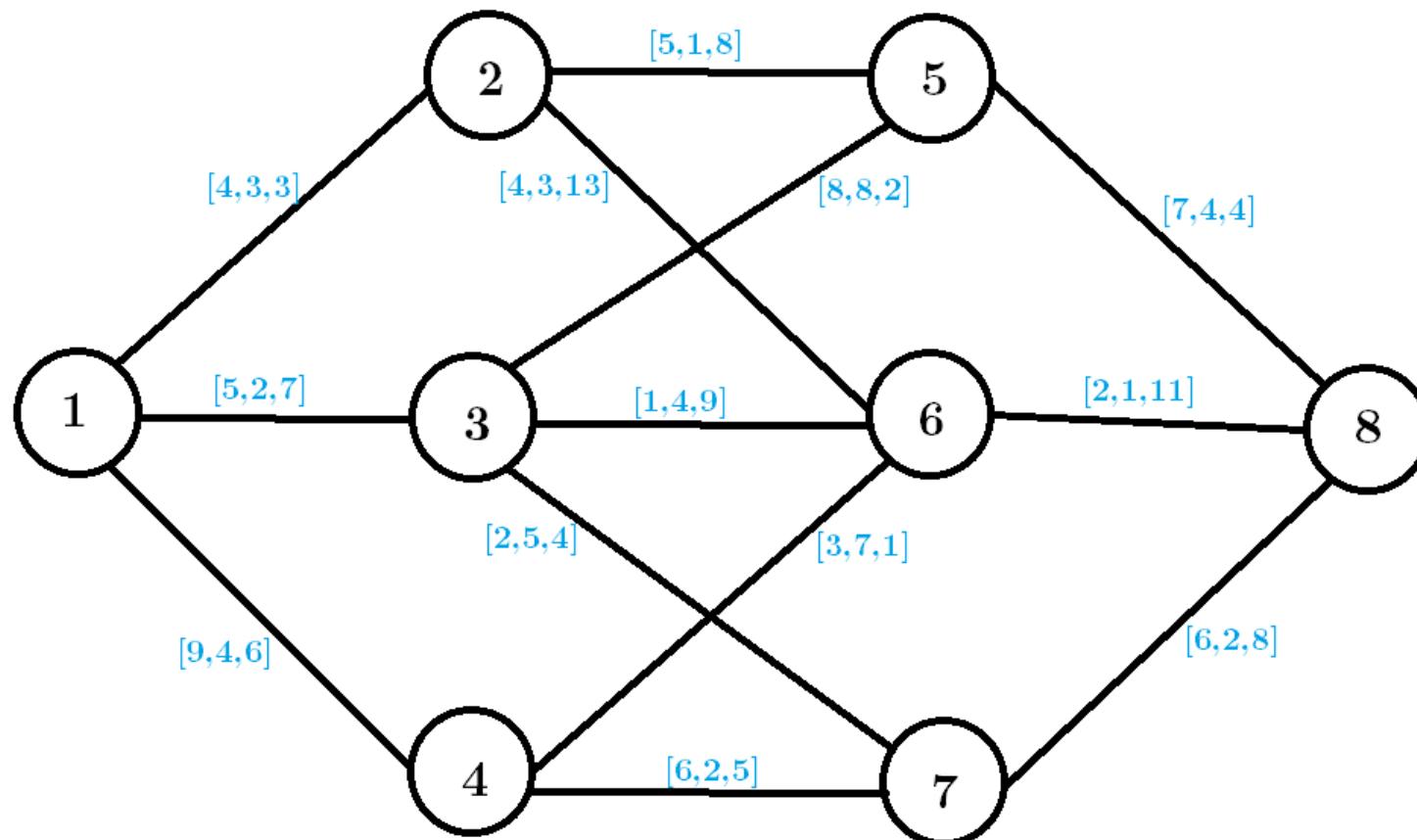
$$\tilde{J}_k^m(x_k) + g_k^m(x_k, u_k) + c^m \leq b^m, \quad m = 2, \ldots, M;$$

- then if obtains $\mathcal{F}_k(x_k)$ by extracting the noninferior subset

- the advantage of using this method is that it allows us to discard as early as possible infeasible solutions, and accordingly reduce the size of the sets $\mathcal{F}_k(x_k)$ and the attendant computation

- additional information, such as a strong upper bound on the optimal cost (from a heuristic) can also be utilized to speed up the computation

- clearly, multiobjective and constrained DP algorithms require quite a bit more computation and storage than ordinary DP for the same system

- it is also possible to use different versions of the label correcting algorithms for multiobjective/constrained shortest path problems – a label of a node would not be just a single number, but rather an M-dimensional vector with components that correspond to the M cost functions

Example 2.12 – Constrained SP

- each path has three "costs": [length, time, fee]

- optimize for length, with budget on time (10) and fee (20)

3 Perfect vs. Imperfect Information

- problems with perfect state information

  – linear systems and quadratic costs

  – scheduling and the interchange argument

- problems with imperfect state information

  – machine repair example

  – Kalman filter

  – linear systems and quadratic costs

- there are quite a bit more examples in the book, be sure to at least skim through them

Problems With Perfect State Information

- we consider applications of discrete-time stochastic optimal control with perfect state information

- these applications are special cases of the basic problem defined in the first chapter and can be addressed via the DP algorithm

- the stochastic nature of the disturbance will be significant – the use of closed-loop control is essential to achieve optimal performance

Example 3.1 – Linear Systems and Quadratic Cost

- special case of a linear system:

$$x_{k+1} = A_k x_k + B_k u_k + w_k, \quad k = 0, 1, \ldots, N-1,$$

and quadratic cost:

$$E_{w_0,\ldots,w_{N-1}}\left\{ x_N' Q_N x_N + \sum_{k=0}^{N-1} (x_k' Q_k x_k + u_k' R_k u_k) \right\},$$

where $x_k \in \mathbb{R}^n$, $u_k \in \mathbb{R}^m$ and the matrices A_k, B_k, Q_k, R_k have appropriate dimensions

- assumptions: Q_k positive semidefinite symmetric, R_k positive definite symmetric, u_k unconstrained, w_k independent random vectors that do not depend on x_k and u_k with zero mean and finite second moment

- popular formulation for a regulation problem where we want to keep the state of the system close to the origin, common paradigm in the theory of automatic control of a motion or a process

- the quadratic cost function is often reasonable as it induces a high penalty for large deviation (both in the state and the controls) and small penalty for small deviations (most importantly, it leads to a nice analytic solution)

- several variation and generalization also have analytic solutions: nonzero mean disturbances w_k, trajectory tracking $(x_i - \bar{x}_i)$, random state transition matrices, etc.

- applying the DP algorithm for the basic problem stated above, we get

$$J_N(x_N) = x_N' Q_N x_N,$$
$$J_k(x_k) = \min_{u_k} E\big\{ x_k' Q_k x_k + u_k' R_k u_k + J_{k+1}(A_k x_k + B_k u_k + w_k) \big\}$$

- it turns out that the cost-to-go functions J_k are quadratic and as a result the optimal control law is a linear function of the state
- to see this, let's compute J_{N-1}:

$$J_{N-1}(x_{N-1}) = \min_{u_{N-1}} E\{x'_{N-1}Q_{N-1}x_{N-1} + u'_{N-1}R_{N-1}u_{N-1}$$

$$+ (A_{N-1}x_{N-1} + B_{N-1}u_{N-1} + w_{N-1})'Q_N(A_{N-1}x_{N-1} + B_{N-1}u_{N-1} + w_{N-1})\}$$

- expanding the last quadratic form and using $E\{w_k\} = 0$, we get the following expression

$$J_{N-1}(x_{N-1}) = x'_{N-1}Q_{N-1}x_{N-1} + \min_{u_{N-1}} \left[u'_{N-1}R_{N-1}u_{N-1} \right.$$

$$+ u_{N-1}B'_{N-1}Q_N B_{N-1}u_{N-1} + 2x'_{N-1}A'_{N-1}Q_N A_{N-1}u_{N-1} \right]$$

$$+ x'_{N-1}A'_{N-1}Q_N A_{N-1}x_{N-1} + E\{w'_{N-1}Q_N w_{N-1}\}$$

- the term $E\{w'Qw\}$ is a quadratic form of a random vector and can be computed as:

$$E\{w'Qw\} = \operatorname{tr}(Q\Sigma) + \mu'Q\mu,$$

where μ is the vector of expected values and Σ is the variance-covariance matrix of the random vector w

- by differentiating w.r.t. u_{N-1} and setting the derivative equal to zero we get

$$(R_{N-1} + B'_{N-1}Q_N B_{N-1})u_{N-1} = -B'_{N-1}Q_N A_{N-1}x_{N-1},$$

where the matrix on the left of u_{N-1} is positive definite (invertible), since R_{N-1} is positive definite and $B'_{N-1}Q_N B_{N-1}$ is positive semidefinite

- the minimizing vector is given by

$$u^*_{N-1} = -(R_{N-1} + B'_{N-1}Q_N B_{N-1})^{-1}B'_{N-1}Q_N A_{N-1}x_{N-1},$$

which we can substitute to the expression for J_{N-1} and get

$$J_{N-1}(x_{N-1}) = x'_{N-1}K_{N-1}x_{N-1} + E\{w'_{N-1}Q_N w_{N-1}\},$$

where

$$K_{N-1} = A'_{N-1}\big(Q_N - Q_N B_{N-1}(B'_{N-1}Q_N B_{N-1}+R_{N-1})^{-1}B'_{N-1}Q_N\big)A_{N-1}+Q_{N-1}$$

141

- the matrix K is positive semidefinite, which implies that J_{N-1} is a positive semidefinite quadratic function (plus an inconsequential constant term), so we may proceed to compute the optimal control law for stage $N-2$

- as before, we show that J_{N-2} is a positive semidefinite quadratic function, and by proceeding sequentially, we obtain the optimal control law for every k:

$$\mu^*(x_k) = L_k x_k,$$

where the gain matrices L_k are given by

$$L_k = -(B_k' K_{k+1} B_k + R_k)^{-1} B_k' K_{k+1} A_k,$$

where the symmetric positive semidefinite matrices K_k are given by recursively the algorithm
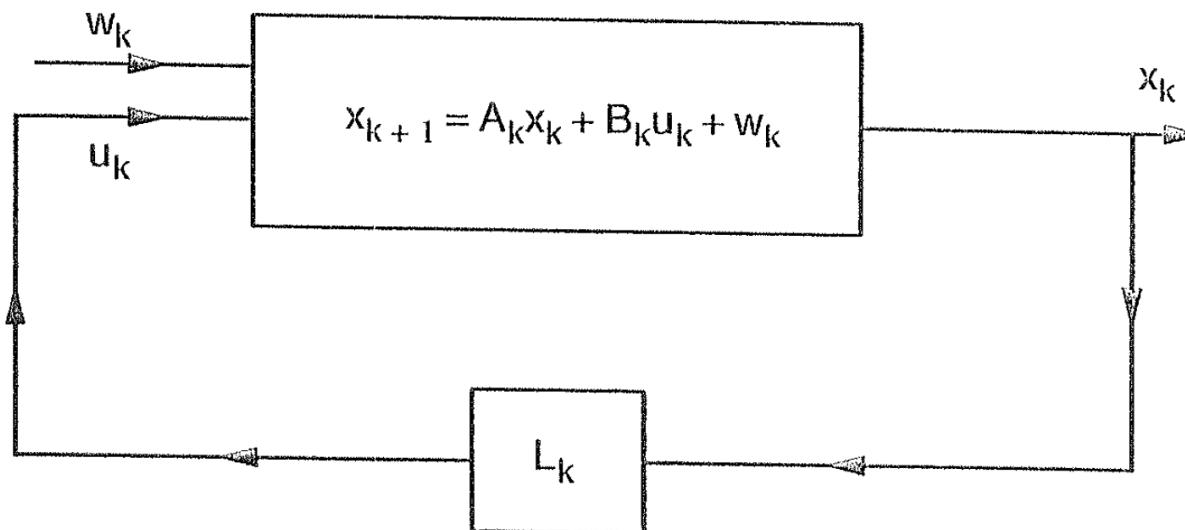
$$K_N = Q_N,$$
$$K_k = A_k'\big(K_{k+1} - K_{k+1} B_k (B_k' K_{k+1} B_k + R_k)^{-1} B_k' K_{k+1}\big) A_k + Q_k$$

- just like DP, this algorithm starts at the terminal time N and proceeds backwards, with the optimal cost given by

$$J_0(x_0) = x_0' K_0 x_0 + \sum_{k=0}^{N-1} E\big\{ w_k' K_{k+1} w_k \big\}$$

- the control law is simple and attractive for engineering applications: the current state x_k is being fed back as input through the linear feedback gain matrix L_k:

- this accounts for the popularity of the LQ formulation

- the recursive equation for the computation of the matrices K_k is called the **discrete-time Ricatti equation,** and has been quite extensively and exhaustively studied
- one interesting property is that if the matrices A_k, B_k, Q_k, R_k are constant and equal to A, B, Q, R, respectively, then the solution K_k converges as $K \to -\infty$ (under mild assumptions) to a steady-state solution K satisfying the algebraic Riccati equation

$$K = A'(K - KB(B'KB + R)^{-1}B'K)A + Q$$

- this property indicates that for the system

$$x_{k+1} = Ax_k + Bu_k + w_k, \quad k = 0, 1, \ldots, N-1,$$

and a large number of stages N, one can reasonably approximate the control law derived earlier by the **stationary** control law $\{\mu^*, \mu^*, \ldots, \mu^*\}$ where

$$\mu^*(x) = Lx,$$
$$L = -(B'KB + R)^{-1}B'KA$$

Example 3.2 – Scheduling and the Interchange Argument

- suppose we are given a collection of tasks to perform but the **ordering** of the tasks is subject to optimal choice

- as examples, consider the ordering of operations in a construction project so as to minimize construction time or the scheduling of jobs in a work-shop so as to minimize machine idle time

- is such problems a useful technique is to start with some schedule and then to interchange two adjacent tasks and see what happens

- first, we show this idea on an example, before formalizing it

The Quiz Problem

- consider a quiz contest where a person is given a list of N questions and can answer these questions in any order he chooses

- question i will be answered correctly with probability p_i, and the person will receive a reward R_i

- at the first incorrect answer, the quiz terminates and the person is allowed to keep his previous rewards

- the problem is to choose the ordering of questions as to maximize the expected reward

- let i and j be the kth and $(k+1)$st questions in an optimally ordered list

$$L = (i_0, \dots, i_{k-1}, i, j, i_{k+2}, \dots, i_{N-1})$$

and consider the list with switched order of i and j

$$L' = (i_0, \dots, i_{k-1}, j, i, i_{k+2}, \dots, i_{N-1})$$

- compare the expected rewards of L and L':

$$E\{\text{reward of } L\} = E\{\text{reward of } \{i_0, \ldots, i_{k-1}\}\}$$
$$+ p_{i_0} \cdots p_{i_{k-1}}(p_i R_i + p_i p_j R_j)$$
$$+ p_{i_0} \cdots p_{i_{k-1}} p_i p_j E\{\text{reward of } \{i_{k+2}, \ldots, i_{N-1}\}\}$$
$$E\{\text{reward of } L'\} = E\{\text{reward of } \{i_0, \ldots, i_{k-1}\}\}$$
$$+ p_{i_0} \cdots p_{i_{k-1}}(p_j R_j + p_j p_i R_i)$$
$$+ p_{i_0} \cdots p_{i_{k-1}} p_i p_j E\{\text{reward of } \{i_{k+2}, \ldots, i_{N-1}\}\}$$

- since L is optimally ordered, we have

$$E\{\text{reward of } L\} \geq E\{\text{reward of } L'\}$$

so it follows that

$$p_i R_i + p_i p_j R_j \geq p_j + p_j p_i R_i \Leftrightarrow \frac{p_i R_i}{1 - p_i} \geq \frac{p_j R_j}{1 - p_j}$$

- to maximize expected reward, questions should be answered in decreasing order of $p_i R_i / (1 - p_i)$

The Interchange Argument

- let us consider the basic problem from the first chapter and formalize the interchange argument used in the previous example

- the main requirement is that the problem has structure such that **there exists an open-loop policy that is optimal**, that is, a sequence of controls that performs as well or better than any sequence of control functions – this is certainly the case for deterministic problems as discussed in the first chapter, but it is also true for some stochastic problems (such as the one above)

- to apply the interchange argument, we start with an optimal sequence

$$\{u_0, \ldots, u_{k-1}, \bar{u}, \tilde{u}, u_{k+2}, \ldots, u_{N-1}\}$$

and focus our attention on the controls $\bar{u}$ and $\tilde{u}$ applied at times k and $k+1$ – we then argue that if the order of $\bar{u}$ and $\tilde{u}$ is interchanged the expected cost cannot decrease

- if X_k is the set of states that can occur with positive probability starting from the given initial state x_0 and using the control subsequence $\{u_0, \ldots, u_{k-1}\}$, we must have for all $x_k \in X_k$

$$
E\{g_k(x_k, \bar{u}, w_k) + g_{k+1}(\bar{x}_{k+1}, \tilde{u}, w_{k+1}) + J^*_{k+2}(\bar{x}_{k+2})\}
$$
$$
\leq E\{g_k(x_k, \tilde{u}, w_k) + g_{k+1}(\tilde{x}_{k+1}, \bar{u}, w_{k+1}) + J^*_{k+2}(\tilde{x}_{k+2})\}
$$

where $\bar{x}_{k+1}$ and $\bar{x}_{k+2}$ (or $\tilde{x}_{k+1}$ and $\tilde{x}_{k+2}$) are the states subsequent to x_k when $u_k = \bar{u}$ and $u_{k+1} = \tilde{u}$ (or $u_k = \tilde{u}$ and $u_{k+1} = \bar{u}$) are applied, and $J^*_{k+2}(\cdot)$ is the optimal cost-to-go function for time $k + 2$

- the relation above is **necessary** condition for optimality and holds for every k and every optimal policy that is open-loop

- there is no guarantee that this necessary condition is powerful enough to lead to an optimal solution, but it is worth considering for specially structured problems

- base of successful heuristics for improving schedules (other problems as well, e.g. maximum independent set)

Problems With Imperfect State Information

- we have assumed so far that the controller has access to the exact value of the current state, but this assumption is often unrealistic

- e.g., some state variables may be inaccessible, the sensor used for measuring them may be inaccurate, or the cost of obtaining the exact value of the state may be prohibitive

- we model situations of this type by assuming that at each stage the controller receives some observations about the value of the current state, which may be corrupted by stochastic uncertainty

- we will find that even though there are DP algorithms for imperfect state information problems, these algorithms are for more computationally demanding – in the absence of analytical solution, imperfect information problems are typically solved suboptimally in practice

Reduction to the Perfect Information Case

Basic Problems with Imperfect State Information

- consider having the basic problem from the first chapter where the controller, instead of having perfect knowledge of the state, has access to observations z_k of the form

$$z_0 = h_0(x_0, v_0), \quad z_k = h_k(x_k, u_{k-1}, v_k), \quad k = 1, 2, \ldots, N-1,$$

  where $z_k \in Z_k$ (observation space), observation disturbance $v_k \in V_k$ with

$$P_{v_k}(\,\cdot\,|\,x_k, \ldots, x_0, u_{k-1}, \ldots, u_0, w_{k-1}, \ldots, w_0, v_{k-1}, \ldots, v_0),$$

  which depends on the current state and the past states, controls, and disturbances

- the initial state x_0 is also a random variable with P_{x_0}

- the probability distribution of w_k may depend on x_k and u_k but not on the prior disturbances $w_0, \ldots, w_{k-1}, v_0, \ldots, v_{k-1}$: $P_{w_k}(\,\cdot\,|\,x_k, u_k)$

- the control u_k is constrained to take values from a given nonempty subset U_k of the control space C_k, which is assumed not to depend on x_k

- let us denote by I_k the information available to the controller at time k and call it the **information vector**, we have

$$I_k = (z_0, z_1, \ldots, z_k, u_0, u_1, \ldots, u_{k-1}), \quad k = 1, 2, \ldots, N-1,$$
$$I_0 = z_0$$

- we consider a class of policies $\pi = \{\mu_0, \mu_1, \ldots, \mu_{N-1}\}$, where each function μ_k maps the information vector I_k into the control space C_k and

$$\mu_k(I_k) \in U_k, \quad \text{for all } I_k, k = 0, 1, \ldots, N-1,$$

and call such policies **admissible**

- we want to find an admissible policy $\pi = \{\mu_0, \mu_1, \ldots, \mu_{N-1}\}$ that minimizes the cost function

$$J_\pi = E_{x_0, w_0, \ldots, w_{N-1}, v_0, \ldots, v_{N-1}} \left\{ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(I_k), w_k) \right\}$$

subject to the system equation

$$x_{k+1} = f_k(x_k, \mu_k(I_k), w_k), \quad k = 0, 1, \ldots, N-1,$$

and the measurement equation

$$z_0 = h_0(x_0, v_0),$$
$$z_k = h_k(x_k, \mu_{k-1}(I_{k-1}), v_k), \quad k = 1, 2, \ldots, N-1$$

- in the perfect information case we were trying to find a control u_k to be applied to every state x_k and time k, now we are looking for a rule that gives the control to be applied for every possible information vector (i.e., for every sequence of observations received and controls employed up to time k)

Reformulation as a Perfect State Information Problem

- the ideas will be similar to the state augmentation techniques – it is intuitively clear that we should define a new system whose state at time k set is set of all variables the knowledge of which can be of benefit to the controller when making the kth decision – the first (and correct) candidate as the state of the new system is the information vector I_k

- from the definition of the information vector

$$I_{k+1} = (I_k, z_{k+1}, u_k), \quad k = 0, 1, \ldots, N - 2, \quad I_0 = z_0,$$

which can be viewed as describing the evolution of a system of the same nature as the one considered in the basic problem

- the state of the system is I_k, the control is u_k, and z_{k+1} can be viewed as a random disturbance

- furthermore, we have

$$P(z_{k+1} \mid I_k, u_k) = P(z_{k+1} \mid I_k, u_k, z_0, z_1, \ldots, z_k),$$

since $z_0, z_1, \ldots, z_k$ are part of the information vector I_k, thus the probability distribution of z_{k+1} depends explicitly only on the state I_k and control u_k of the new system and not on the prior "disturbances" $z_k, \ldots, z_0$

- by writing

$$E\{g_k(x_k, u_k, w_k)\} = E\Big\{ E_{x_k, w_k}\{g_k(x_k, u_k, w_k) \mid I_k, u_k\}\Big\}$$

we can similarly reformulate the cost function in terms of the variables of the new system

- the cost per stage as a function of the new state I_k and the control u_k is

$$\tilde{g}_k(I_k, u_k) = E_{x_k, w_k}\{g_k(x_k, u_k, w_k) \mid I_k, u_k\}$$

- thus, the basic problem with imperfect state information has been refor-mulated as a problem with perfect state information that involves the I_k system with appropriate cost per stage

- by writing the DP algorithm for this latter problem as

$$J_{N-1}(I_{N-1}) = \min_{u_{N-1} \in U_{N-1}} \left[E_{x_{N-1}, w_{N-1}} \big\{ g_N(f_{N-1}(x_{N-1}, u_{N-1}, w_{N-1})) \right.$$
$$\left. + g_{N-1}(x_{N-1}, u_{N-1}, w_{N-1}) \,|\, I_{N-1}, u_{N-1} \big\} \right],$$

and for $k = 0, 1, \ldots, N-2$,

$$J_k(I_k) = \min_{u_k \in U_k} \left[E_{x_k, w_k, z_{k+1}} \big\{ g_k(x_k, u_k, w_k) + J_{k+1}(I_k, z_{k+1}, u_k) \,|\, I_k, u_k \big\} \right]$$

- these equations constitute one possible DP algorithm for the imperfect state information problem

- an optimal policy $\{\mu_0^*, \mu_1^*, \ldots, \mu_{N-1}^*\}$ is obtained by minimizing in the right-hand size for every possible value of the information vector I_{N-1} to obtain $\mu_{N-1}(I_{N-1})$

- simultaneously, $J_{N-1}(I_{N-1})$ is computed and used in the computation of $J_{N-2}(I_{N-2})$ via the minimization in the DP equation, which is carried out for every possible value I_{N-2}

- proceeding with $J_{N-3}(I_{N-3})$ and μ_{N-3}^* and so on, until $J_0(I_0) = J_0(z_0)$ is computed

- the optimal cost J^* is then given by
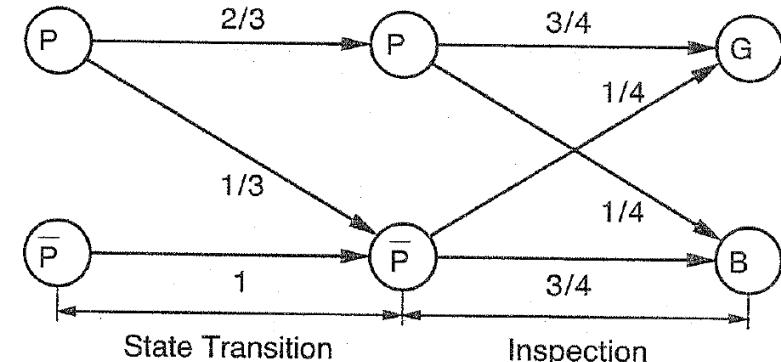
$$J^* = E_{z_0}\{J_0(z_0)\}$$

Example 3.3 – Machine Repair

- consider a machine that can be in two states: proper condition (good state) P and improper condition (bad state) $\bar{P}$

- if the machine is operated for one time period, it stays in state P with probability $\frac{2}{3}$ if it started in P, and it stays in $\bar{P}$ with probability 1 if it started in $\bar{P}$

- at the end of the first and second time periods the machine is inspected and there are two possible inspection outcomes: G (probably good state) and B (probably bad state)

- if the machine is in a good state P, the inspection outcome is G with probability $\frac{3}{4}$; if the machine is in the bad state $\bar{P}$, the inspection outcome is B with probability $\frac{3}{4}$

$$P(G \mid x = P) = \frac{3}{4}, \ P(B \mid x = P) = \frac{1}{4}$$

$$P(G \mid x = \bar{P}) = \frac{1}{4}, \ P(B \mid x = \bar{P}) = \frac{3}{4}$$

State Transition   Inspection

- after each inspection one of two possible actions can be taken:

  C: continue operation

  S: stop the machine, determine its state through accurate diagnostics, and if it is in the bad state $\bar{P}$ bring it back to the good state P

- the problem is to determine the policy that minimizes the expected costs over the three time periods – we want find the optimal action after the result of the first inspection is known, and after the results of the first and second inspections, as well as the action taken after the first inspection, are known

159

- the state space consists of the two states

$$\text{state space } = \{P, \bar{P}\},$$

  and the control space consists of the two actions

$$\text{control space } = \{C, S\}$$

- the system evolution my be described by introducing the system equation

$$x_{k+1} = w_k, \quad k = 0, 1,$$

  where the probability distribution of w_k is given by

$$P(w_k = P \mid x_k = P, u_k = C) = \frac{2}{3}, \quad P(w_k = \bar{P} \mid x_k = P, u_k = C) = \frac{1}{3}$$

$$P(w_k = P \mid x_k = \bar{P}, u_k = C) = 0, \quad P(w_k = \bar{P} \mid x_k = \bar{P}, u_k = C) = 1$$

$$P(w_k = P \mid x_k = P, u_k = S) = \frac{2}{3}, \quad P(w_k = \bar{P} \mid x_k = P, u_k = S) = \frac{1}{3}$$

$$P(w_k = P \mid x_k = \bar{P}, u_k = S) = \frac{2}{3}, \quad P(w_k = \bar{P} \mid x_k = \bar{P}, u_k = S) = \frac{1}{3}$$

- we denote by x_0, x_1, x_2 the state of the machine at the end of the first, second, and third time period, and by u_0 the action taken after the first inspection (end of first time period) and by u_1 the action taken after the second inspection (end of second time period)

- the probability distribution of x_0 is: $P(x_0 = P) = \frac{2}{3}, \ P(x_0 = \bar{P}) = \frac{1}{3}$

- note that we do not have perfect state information, since the inspection do not reveal the state of the machine with certainty – rather the result of each inspection may be viewed as a measurement of the system state of the form
$$z_k = v_k, \quad k = 0, 1,$$

where for $k = 0, 1$, the probability distribution of v_k is given by
$$P(v_k = G \,|\, x_k = P) = \frac{3}{4}, \ P(v_k = B \,|\, x_k = P) = \frac{1}{4}$$
$$P(v_k = G \,|\, x_k = \bar{P}) = \frac{1}{4}, \ P(v_k = B \,|\, x_k = \bar{P}) = \frac{3}{4}$$

- the cost resulting from the sequence of states x_0, x_1 and actions u_0, u_1 is

$$g(x_0, u_0) + g(x_1, u_1),$$

where

$$g(P, C) = 0, \ g(P, S) = 1, \ g(\bar{P}, C) = 2, \ g(\bar{P}, S) = 1$$

- the information vector at times 0 and 1 is

$$I_0 = z_0, \quad I_1 = (z_0, z_1, u_0),$$

and we seek functions $\mu_0(I_0)$, $\mu_1(I_1)$ that minimize

$$E_{x_0, w_0, w_1; v_0, v_1} \left\{ g(x_0, \mu_0(I_0)) + g(x_1, \mu_1(I_1)) \right\}$$

$$= E_{x_0, w_0, w_1; v_0, v_1} \left\{ g(x_0, \mu_0(z_0)) + g(x_1, \mu_1(z_0, z_1, \mu_0(z_0))) \right\}$$

- now we apply the DP algorithm – it involves taking the minimum of two possible actions (C and S) and it has the form

$$J_k(I_k) = \min \big[P(x_k = P \,|\, I_k, C) g(P, C) + P(x_k = \bar{P} \,|\, I_k, C) g(\bar{P}, C)$$

$$+ E_{z_{k+1}} \{ J_{k+1}(I_{k+1}, C, z_{k+1}) \,|\, I_k, C \},$$

$$P(x_k = P \,|\, I_k, S) g(P, S) + P(x_k = \bar{P} \,|\, I_k, S) g(\bar{P}, S)$$

$$+ E_{z_{k+1}} \{ J_{k+1}(I_{k+1}, S, z_{k+1}) \,|\, I_k, S \} \big],$$

where $k = 0, 1$, and the terminal condition $J_2(I_2) = 0$

- *Last stage*: We compute $J_1(I_1)$ for each of the eight possible information vectors $I_1 = (z_0, z_1, u_1)$. As indicated by the above DP algorithm, for each of these vectors, we shall compute the expected cost of possible actions, $u_1 = C$ and $u_1 = S$, and select as optimal the one with the smallest cost

- we have: cost of $C = 2 \cdot P(x_1 = \bar{P} \,|\, I_1)$,   cost of $S = 1$, and therefore

$$J_1(I_1) = \min[2P(x_1 = \bar{P} \,|\, I_1), 1]$$

- the probabilities $P(x_1 = \bar{P} \mid I_1)$ can be computed using Bayes' rule and the problem data:

(1) For $I_1 = (G, G, S)$:

$$P(x_1 = \bar{P} \mid G, G, S) = \frac{P(x_1 = \bar{P}, G, G \mid S)}{P(G, G \mid S)} = \frac{\frac{1}{3} \cdot \left(\frac{2}{3} \cdot \frac{3}{4} + \frac{1}{3} \cdot \frac{1}{4}\right) \cdot \frac{1}{4}}{\left(\frac{2}{3} \cdot \frac{3}{4} + \frac{1}{3} \cdot \frac{1}{4}\right)^2} = \frac{1}{7}$$

hence: $\quad J_1(G, G, S) = \frac{2}{7}, \quad \mu_1^*(G, G, S) = C$

(2) For $I_1 = (B, G, S)$:

$$P(x_1 = \bar{P} \mid B, G, S) = P(x_1 = \bar{P} \mid G, G, S) = \frac{1}{7}$$

hence: $\quad J_1(B, G, S) = \frac{2}{7}, \quad \mu_1^*(B, G, S) = C$

(3) For $I_1 = (G, B, S)$:

$$P(x_1 = \bar{P} \mid G, B, S) = \frac{P(x_1 = \bar{P}, G, B \mid S)}{P(G, B \mid S)} = \frac{\frac{1}{3} \cdot \left(\frac{2}{3} \cdot \frac{3}{4} + \frac{1}{3} \cdot \frac{1}{4}\right) \cdot \frac{3}{4}}{\left(\frac{2}{3} \cdot \frac{1}{4} + \frac{1}{3} \cdot \frac{3}{4}\right)\left(\frac{2}{3} \cdot \frac{3}{4} + \frac{1}{3} \cdot \frac{1}{4}\right)} = \frac{3}{5}$$

hence: $\quad J_1(G, B, S) = 1, \quad \mu_1^*(G, B, S) = S$

(4) For $I_1 = (B, B, S)$:

$$P(x_1 = \bar{P} \mid B, B, S) = P(x_1 = \bar{P} \mid G, B, S) = \frac{3}{5}$$

hence: $\quad J_1(B, B, S) = 1, \quad \mu_1^*(B, B, S) = S$

- cont'd:

(5) For $I_1 = (G, G, C)$:

$$P(x_1 = \bar{P} \mid G, G, C) = \frac{P(x_1 = \bar{P}, G, G \mid C)}{P(G, G \mid C)} = \frac{1}{5}$$

hence: $J_1(G, G, C) = \frac{2}{5}, \quad \mu_1^*(G, G, C) = C$

(6) For $I_1 = (B, G, C)$:

$$P(x_1 = \bar{P} \mid B, G, C) = \frac{11}{23}$$

hence: $J_1(B, G, C) = \frac{22}{23}, \quad \mu_1^*(B, G, C) = C$

(7) For $I_1 = (G, B, C)$:

$$P(x_1 = \bar{P} \mid G, B, C) = \frac{9}{13}$$

hence: $J_1(G, B, C) = 1, \quad \mu_1^*(G, B, C) = S$

(8) For $I_1 = (B, B, C)$:

$$P(x_1 = \bar{P} \mid B, B, C) = \frac{33}{37}$$

hence: $J_1(B, B, C) = 1, \quad \mu_1^*(B, B, C) = S$

- to summarize: the optimal policy is to continue $(u_1 = C)$ if the result of the last inspection was G, and to stop $(u_1 = S)$ if the results of the last inspection was B

- *First stage*: We compute $J_0(I_0)$ for the two possible information vectors $I_0 = (G)$, $I_0 = (B)$

- we have:

$$\text{cost of } C = 2P(x_0 = \bar{P} \,|\, I_0, C) + E_{z_1}\{J_1(I_0, z_1, C) \,|\, I_0, C\}$$
$$= 2P(x_0 = \bar{P} \,|\, I_0, C) + P(z_1 = G \,|\, I_0, C)J_1(I_0, G, C)$$
$$+ P(z_1 = B \,|\, I_0, C)J_1(I_0, B, C)$$
$$\text{cost of } S = 1 + E_{z_1}\{J_1(I_0, z_1, S) \,|\, I_0, S\}$$
$$= 1 + P(z_1 = G \,|\, I_0, S)J_1(I_0, G, S) + P(z_1 = B \,|\, I_0, S)J_1(I_0, B, S)$$

and

$$J_0(I_0) = \min\big[2P(x_0 = \bar{P} \,|\, I_0, C) + E_{z_1}\{J_1(I_0, z_1, C) \,|\, I_0, C\},$$
$$1 + E_{z_1}\{J_1(I_0, z_1, S) \,|\, I_0, S\}\big]$$

(1) For $I_0 = (G)$ we get:

$$P(z_1 = G \mid G, C) = \frac{15}{28}, \quad P(z_1 = B \mid G, C) = \frac{13}{28}$$

$$P(z_1 = G \mid G, S) = \frac{7}{12}, \quad P(z_1 = B \mid G, S) = \frac{5}{12}$$

$$P(x_0 = \bar{P} \mid G, C) = \frac{1}{7}$$

and hence

$$J_0(G) = \min \left[2 \cdot \frac{1}{7} + \frac{15}{28} J_1(G, G, C) + \frac{13}{28} J_1(G, B, C), \right.$$

$$\left. 1 + \frac{7}{12} J_1(G, G, S) + \frac{5}{12} J_1(G, B, S) \right]$$

and using values of J_1 from the previous stage

$$J_0(G) = \min \left[2 \cdot \frac{1}{7} + \frac{15}{28} \cdot \frac{2}{5} + \frac{13}{28} \cdot 1, 1 + \frac{7}{12} \cdot \frac{2}{7} + \frac{5}{12} \cdot 1 \right]$$

$$= \min \left[\frac{27}{28}, \frac{19}{12} \right]$$

$$J_0(G) = \frac{27}{28}, \quad \mu_0^*(G) = C$$

(2) For $I_0 = (B)$ we get:
$$P(z_1 = G \mid B, C) = \frac{23}{60}, \quad P(z_1 = B \mid B, C) = \frac{37}{60}$$

$$P(z_1 = G \mid B, S) = \frac{7}{12}, \quad P(z_1 = B \mid B, S) = \frac{5}{12}$$

$$P(x_0 = \bar{P} \mid B, C) = \frac{3}{5}$$

and hence
$$J_0(B) = \min \Big[2 \cdot \frac{3}{5} + \frac{23}{60} J_1(B, G, C) + \frac{37}{60} J_1(B, B, C),$$

$$1 + \frac{7}{12} J_1(B, G, S) + \frac{5}{12} J_1(B, B, S) \Big]$$

and using values of J_1 from the previous stage
$$J_0(B) = \min \Big[\frac{131}{60}, \frac{19}{12} \Big] = \frac{19}{12}$$

$$J_0(G) = \frac{19}{12}, \quad \mu_0^*(B) = S$$

168

- to summarize: the optimal policy for both stages is to continue if the result of the latest inspection is G, and to stop and repair otherwise

- the optimal cost is

$$J^* = P(G)J_0(G) + P(B)J_0(B)$$

which, using $P(G) = \frac{7}{12}$ and $P(B) = \frac{5}{12}$, yields

$$J^* = \frac{7}{12} \cdot \frac{27}{28} + \frac{5}{12} \cdot \frac{19}{12} = \frac{176}{144}$$

- in this example, the computation of the optimal policy and cost by the DP algorithm was possible mainly because the example was very simple

- it is easy to see that for more complex problems, the computational requirements skyrocket very fast; this makes the application of the algorithm very difficult (or impossible) in many cases

- however, there are some problems for which analytical solution is possible

Kalman Filtering

- least-squares estimation with application in estimating the state of linear discrete-time dynamic system

- there are two random vectors x and y, which are related through their joint probability distribution so that each value of one provides information about the value of the other

- we get to know the value of y, and we want to estimate the value of x so that the average squared error between x and its estimate is minimized

- a related problem is to find the best estimate of x within the class of all estimates that are **linear** in the measured vector y

- we specialize these problems to a case where there is an underlying linear dynamic system

Least-Square Estimation

- consider two jointly distributed random vectors x and y taking values in $\mathbb{R}^n$ and $\mathbb{R}^m$

- we view y as a measurement that provides some information about x
  – while prior knowing y our estimate of x may have been the expected value $E\{x\}$, once the value of y is known, we want to form an update estimate $x(y)$ of the value x

- this updated estimate depends on y, so we are interested in a rule that gives us the estimate for each possible value of y, i.e., we are interested in a function $(\,\cdot\,)$, where $x(y)$ is the estimate of x given y

- such function

$$x(\,\cdot\,) \ : \ \mathbb{R}^m \to \mathbb{R}^n$$

  is called an **estimator**

- we are seeking an estimator that is optimal in the following sense – it is a minimum of the following criterion

$$E_{x,y}\{||x - x(y)||^2\},$$

  and assume that all expected values are finite

- an estimator that minimizes the expected squared error above over all $x(\,\cdot\,)$ is called a **least-squared estimator** and is denoted by $x^*(\,\cdot\,)$

- since

$$E_{x,y}\{||x - x(y)||^2\} = E_y\big\{E_x\{||x - x(y)||^2 \,|\, y\}\big\},$$

  it is clear that $x^*(\,\cdot\,)$ is a least-square estimator if $x^*(y)$ minimizes the conditional expectation in the right-hand side above for every $y \in \mathbb{R}^m$, that is

$$E_x\big\{||x - x(y)||^2 \,|\, y\big\} = \min_{z \in \mathbb{R}^m} E_x\{||x - z||^2 \,|\, y\}, \quad \text{for all } y \in \mathbb{R}^m$$

- for every fixed $z \in \mathbb{R}^n$

$$E_x\{||x - z||^2 \,|\, y\} = E_x\{||x||^2 \,|\, y\} - 2z'E_x\{x \,|\, y\} + ||z||^2$$

by setting to zero the derivative with respect to z, we see that the above expression minimizes by $z = E_x\{x \,|\, y\}$, i.e. $x^*(y) = E_x\{x \,|\, y\}$

- this estimator may be a complicated nonlinear function of y (making the calculation difficult), which motivates finding optimal estimators within the restricted class of **linear** estimators

$$x(y) = Ay + b,$$

where A is a $n \times m$ matrix and b is an n-dimensional vector

- an estimator
$$\hat{x}(y) = \hat{A}y + \hat{b},$$
where $\hat{A}$ and $\hat{b}$ minimize, $E_{x,y}\{||x - Ay - b||^2\}$ over all $n \times m$ matrices A and vectors $b \in \mathbb{R}^n$ is called a **linear least-squares estimator**

- in the special case where x and y are jointly Gaussian random vectors it turns out that the conditional expectation $E_x\{x \mid y\}$ is a linear function of y (plus a constant vector), and as a results, a linear least-square estimator is also a least-square estimator

- [E.3] let x, y be random vectors taking values in $\mathbb{R}^n$ and $\mathbb{R}^m$, with given joint probability distribution, with expected values and covariance matrices of x, y denoted by

$$E\{x\} = \bar{x}, \quad E\{y\} = \bar{y},$$
$$E\{(x - \bar{x})(x - \bar{x})'\} = \Sigma_{xx}, \quad E\{(y - \bar{y})(y - \bar{y})'\} = \Sigma_{yy},$$
$$E\{(x - \bar{x})(y - \bar{y})'\} = \Sigma_{xy}, \quad E\{(y - \bar{y})(x - \bar{x})'\} = \Sigma_{yx},$$

and assume Σ_{yy} is invertible

- then the linear least-squares estimator of x given y is

$$\hat{x}(y) = \bar{x} + \Sigma_{xy}\Sigma_{yy}^{-1}(y - \bar{y})$$

- the corresponding error covariance matrix is given by

$$E_{x,y} = \left\{ (x - \hat{x}(y))(x - \hat{x}(y))' \right\} = \Sigma_{xx} - \Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{yx}$$

- with the following properties

  - [E.3.1] the linear least-squares estimator is unbiased, i.e.

  $$E_y\{\hat{x}(y)\} = \bar{x},$$

  - [E.3.2] the estimation error $x - \hat{x}(y)$ is uncorrelated with both y and $\hat{x}(y)$ (orthogonal projection principle), i.e.

  $$E_{x,y}\left\{ y(x - \hat{x}(y))' \right\} = 0,$$
  $$E_{x,y}\left\{ \hat{x}(y)(x - \hat{x}(y))' \right\} = 0,$$

- [E.3.3] consider in addition to x and y, the random vector z defined by

$$z = Cx,$$

where C is a given $p \times n$ matrix, then the linear least-squares estimate of z given y is

$$\hat{z}(y) = C\hat{x}(y),$$

and the corresponding error covariance matrix is given by

$$E_{x,y}\Big\{ (z - \hat{z}(y))(z - \hat{z}(y))' \Big\} = C E_{x,y}\Big\{ (x - \hat{x}(y))(x - \hat{x}(y))' \Big\} C'$$

- [E.3.4] consider in addition to x and y, the random vector z defined by

$$z = Cy + u,$$

where C is a given $p \times m$ matrix of rank p and u is a given vector in $\mathbb{R}^p$, then the linear least-square estimate $\hat{x}(z)$ of x given z is

$$\hat{x}(z) = \bar{x} + \Sigma_{xy}C'(C\Sigma_{yy}C')^{-1}(z - C\bar{y} - u),$$

and the corresponding error covariance matrix is

$$E_{x,z}\Big\{ (x - \hat{x}(z))(x - \hat{x}(z))' \Big\} = \Sigma_{xx} - \Sigma_{xy}C'(C\Sigma_{yy}C')^{-1}C\Sigma_{yx}$$

- frequently we want to estimate a vector of parameters $x \in \mathbb{R}^n$ given a measurement vector $z \in \mathbb{R}^m$ of the form $z = Cx + v$, where C is a given matrix and $v \in \mathbb{R}^m$ is a random measurement error

- [E.3.5] let

$$z = Cx + v$$

where C is a given $m \times n$ matrix, and the random vectors $x \in \mathbb{R}^n$, $v \in \mathbb{R}^m$ are uncorrelated and denote

$$E\{x\} = \bar{x}, \quad E\{(x - \bar{x})(x - \bar{x})'\} = \Sigma_{xx}$$
$$E\{v\} = \bar{v}, \quad E\{(v - \bar{v})(v - \bar{v})'\} = \Sigma_{vv}$$

and assume further that Σ_{vv} is positive definite, then

$$\hat{x}(z) = \bar{x} + \Sigma_{xx}C'(C\Sigma_{xx}C' + \Sigma_{vv})^{-1}(z - C\bar{x} - \bar{v}),$$
$$E_{x,v}\left\{(x - \hat{x}(z))(x - \hat{x}(z))'\right\} = \Sigma_{xx} - \Sigma_{xx}C'(C\Sigma_{xx}C' + \Sigma_{vv})^{-1}C\Sigma_{xx}$$

- next, we deal with estimates involving multiple measurements obtained sequentially, and show how to modify an existing estimate $\hat{x}(y)$ to obtain $\hat{x}(y, z)$ once an additional vector z becomes known

- [E.3.6] consider in addition to x and y, an additional random vector z taking values in $\mathbb{R}^p$, which is uncorrelated with y. Then the linear least-squares estimate $\hat{x}(y, z)$ of x given y and z has the form

$$\hat{x}(y, z) = \hat{x}(y) + \hat{x}(z) - \bar{x},$$

  where $\hat{x}(y)$ and $\hat{x}(z)$ are the linear least-squares estimates of x given y and z, and

$$E_{x,y,z}\left\{(x - \hat{x}(y, z))(x - \hat{x}(y, z))'\right\} = \Sigma_{xx} - \Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{yx} - \Sigma_{xz}\Sigma_{zz}^{-1}\Sigma_{zx},$$

  where

$$\Sigma_{xz} = E_{x,z}\left\{(x - \bar{x})(z - \bar{z})'\right\}, \quad \Sigma_{zx} = E_{x,z}\left\{(z - \bar{z})(x - \bar{x})'\right\},$$
$$\Sigma_{zz} = E_z\left\{(z - \bar{z})(z - \bar{z})'\right\}, \quad \bar{z} = E_z\{z\},$$

  and it is assumed that Σ_{zz} is invertible

- [E.3.7] let z be as on the previous slide and assume that y and z are not necessarily uncorrelated, that is, we may have

$$\Sigma_{yz} = \Sigma'_{zy} = E_{y,z}\big\{(y - \bar{y})(z - \bar{z})'\big\} \neq 0,$$

then

$$\hat{x}(y, z) = \hat{x}(y) + \hat{x}(z - \hat{z}(y)) - \bar{x},$$

where $\hat{x}(z - \hat{z}(y))$ denotes the linear least-squares estimate of x given the random vector $z - \hat{z}(y)$ and $\hat{z}(y)$ is the linear least-squares estimate of z given y, and we have

$$E_{x,y,z}\big\{(x - \hat{x}(y, z))(x - \hat{x}(y, z))'\big\} = \Sigma_{xx} - \Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{yx} - \hat{\Sigma}_{xz}\hat{\Sigma}_{zz}^{-1}\hat{\Sigma}_{xz},$$

where

$$\hat{\Sigma}_{xz} = E_{x,y,z}\big\{(x - \bar{x})(z - \hat{z}(y))'\big\}$$

$$\hat{\Sigma}_{zz} = E_{y,z}\big\{(z - \hat{z}(y))(z - \hat{z}(y))'\big\}$$

$$\hat{\Sigma}_{zx} = E_{x,y,z}\big\{(z - \hat{z}(y))(x - \bar{x})'\big\}$$

179

State Estimation – The Kalman Filter

- consider a linear dynamic system but without a control vector ($u_k = 0$)

$$x_{k+1} = A_k x_k + w_k, \quad k = 0, 1, \ldots, N - 1, \quad \text{(E.30)}$$

  where the state is $x_k \in \mathbb{R}^n$, disturbance is $w_k \in \mathbb{R}^m$, and the matrices A_k are known

- consider also the measurement equation

$$z_k = C_k x_k + v_k, \quad k = 0, 1, \ldots, N - 1, \quad \text{(E.31)}$$

  where $z_k \in \mathbb{R}^s$ is the observation, and $v_k \in \mathbb{R}^s$ is the observation noise

- assume that $x_0, w_0, \ldots, w_{N-1}, v_0, \ldots, v_{N-1}$ are independent vectors with given probability distributions and that

$$E\{w_k\} = E\{v_k\} = 0, \quad k = 0, 1, \ldots, N - 1 \quad \text{(E.32)}$$

  and use the notation

$$S = E\Big\{ (x_0 - E\{x_0\})(x_0 - E\{x_0\})' \Big\}, \quad M_k = E\{w_k w_k'\}, \quad N_k = E\{v_k v_k'\}$$

- we first show a straightforward (but tedious) method to derive the linear least-squares estimate of x_{k+1} or x_k given the values $z_0, z_1, \ldots, z_k$, denote

$$Z_k = (z_0', z_1', \ldots, z_k')', \quad r_{k-1} = (x_0', w_0', w_1', \ldots, w_{k-1}')'$$

- find the linear least-square estimate of r_{k-1} given Z_k, and obtain the linear least-square estimate of x_k given Z_k after expressing x_k as a linear function of r_{k-1}

- for each i with $0 \le i \le k$ we have, by using the system equation

$$x_{i+1} = L_i r_i,$$

where L_i is the $n \times (n(i+1))$ matrix

$$L_i = (A_i \cdots A_0, A_i \cdots A_1, \ldots, A_i, I)$$

- as a result we may write $Z_k = \Phi_{k-1} r_{k-1} + V_k$, where $V_k = (v_0', v_1', \ldots, v_k')'$, and Φ_{k-1} is an $s(k+1) \times (nk)$ matrix of the form

$$\Phi_{k-1} = \begin{pmatrix} C_0 & 0 \\ C_1 L_0 & 0 \\ \vdots & \vdots \\ C_{k-1} L_{k-2} & 0 \\ C_k L_{k-1} \end{pmatrix}$$

- we can use [E.3.5], the equations above, and the data of the problem to compute

$$\hat{r}_{k-1}(Z_k) \quad \text{and} \quad E\Big\{ (r_{k-1} - \hat{r}_{k-1}(Z_k))(r_{k-1} - \hat{r}_{k-1}(Z_k))' \Big\}$$

- denote the linear least-squares estimate of x_{k+1} and x_k given Z_k by $\hat{x}_{k+1\,|\,k}$ and $\hat{x}_{k\,|\,k}$, we can obtain $\hat{x}_{k\,|\,k} = \hat{x}_k(Z_k)$ from (and $\hat{x}_{k+1\,|\,k}$) [E.3.3]

$$\hat{x}_{k\,|\,k} = L_{k-1} \hat{r}_{k-1}(Z_k),$$

$$E\Big\{ (x_k - \hat{x}_{k\,|\,k})(x_k - \hat{x}_{k\,|\,k})' \Big\} = L_{k-1} E\Big\{ (r_{k-1} - \hat{r}_{k-1}(Z_k))(r_{k-1} - \hat{r}_{k-1}(Z_k))' \Big\} L_{k-1}'$$

- the preceding method for obtaining the least-squares estimate of x_k is cumbersome when the number of measurements is large

- fortunately, the sequential structure of the problem can be exploited and the computations can be organized conveniently

- the main attractive feature of the Kalman filtering algorithm is that the estimate $\hat{x}_{k+1 \mid k}$ can be obtained by means of a simple equation that involves the previous estimate $\hat{x}_{k \mid k-1}$ and the new measurement z_k, but **does not involve any of the past measurements** $z_0, z_1, \dots, z_{k-1}$

- suppose that we have computed the estimate $\hat{x}_{k \mid k-1}$ together with the covariance matrix

$$\Sigma_{k \mid k-1} = E\left\{(x_k - \hat{x}_{k \mid k-1})(x_k - \hat{x}_{k \mid k-1})'\right\} \qquad \text{(E.34)}$$

- at time k we receive the additional measurement

$$z_k = C_k x_k + v_k$$

183

- we may now use [E.3.7] to compute the linear least-squares estimate of x_k given $Z_{k-1} = (z_0', z_1', \ldots, z_{k-1}')$ and z_k, which is denoted by $\hat{x}_{k\,|\,k}$ and given by

$$\hat{x}_{k\,|\,k} = \hat{x}_{k\,|\,k-1} + \hat{x}_k(z_k - \hat{z}_k(Z_{k-1})) - E\{x_k\}, \qquad \text{(E.35)}$$

where $\hat{z}_k(Z_{k-1})$ is the linear least-squares estimate of z_k given Z_{k-1} and $\hat{x}_k(z_k - \hat{z}_k(Z_{k-1}))$ is the linear least-squares estimate of x_k given $(z_k - \hat{z}_k(Z_{k-1}))$

- first we calculate the term $\hat{x}_k(z_k - \hat{z}_k(Z_{k-1}))$:

$$\hat{z}_k(Z_{k-1}) = C_k \hat{x}_{k\,|\,k-1}, \qquad \text{(E.36)}$$

$$E\Big\{ (z_k - \hat{z}_k(Z_{k-1}))(z_k - \hat{z}_k(Z_{k-1}))' \Big\} = C_k \Sigma_{k\,|\,k-1} C_k' + N_k, \qquad \text{(E.37)}$$

$$E\Big\{ x_k(z_k - \hat{z}_k(Z_{k-1}))' \Big\} = E\Big\{ x_k(C_k(x_k - \hat{x}_{k\,|\,k-1}))' \Big\} + E\{x_k v_k'\}$$

$$= E\Big\{ (x_k - \hat{x}_{k\,|\,k-1})(x_k - \hat{x}_{k\,|\,k-1})' \Big\} C_k' + E\Big\{ \hat{x}_{k\,|\,k-1}(x_k - \hat{x}_{k\,|\,k-1})' \Big\} C_k'$$

- the last term in the rhs above is zero from [E.3.2], meaning

$$E\Big\{ x_k(z_k - \hat{z}_k(Z_{k-1}))' \Big\} = \Sigma_{k\,|\,k-1} C_k' \qquad \text{(E.38)}$$

- using (E.36-E38) in [E.3] we get

$$\hat{x}_k(z_k - \hat{z}_k(Z_{k-1})) = E\{x_k\} + \Sigma_{k\,|\,k-1}C_k'(C_k\Sigma_{k\,|\,k-1}C_k' + N_k)^{-1}(z_k - C_k\hat{x}_{k\,|\,k-1})$$

- we can also write (E.35) as

$$\hat{x}_{k\,|\,k} = \hat{x}_{k\,|\,k-1} + \Sigma_{k\,|\,k-1}C_k'(C_k\Sigma_{k\,|\,k-1}C_k' + N_k)^{-1}(z_k - C_k\hat{x}_{k\,|\,k-1}) \qquad (\text{E.39})$$

- by using [E.3.3] we have

$$\hat{x}_{k+1\,|\,k} = A_k\hat{x}_{k\,|\,k} \qquad (\text{E.40})$$

- concerning the covariance matrix $\Sigma_{k+1\,|\,k}$, we have from (E.30), (E.32), (E.33) and [E.3.3]

$$\Sigma_{k+1\,|\,k} = A_k\Sigma_{k\,|\,k}A_k' + M_k, \qquad (\text{E.41})$$

where

$$\Sigma_{k\,|\,k} = E\Big\{(x_k - \hat{x}_{k\,|\,k})(x_k - \hat{x}_{k\,|\,k})'\Big\}$$

185

- the error covariance matrix $\Sigma_{k\,|\,k}$ may be computed via [E.3.7] similar to $\hat{x}_{k\,|\,k}$ as

$$\Sigma_{k\,|\,k} = \Sigma_{k\,|\,k-1} - \Sigma_{k\,|\,k-1}C_k'(C_k\Sigma_{k\,|\,k-1}C_k' + N_k)^{-1}C_k\Sigma_{k\,|\,k-1} \qquad (E.42)$$

- equations (E.39-E.42) with the initial conditions

$$\hat{x}_{0\,|\,-1} = E\{x_0\}, \quad \Sigma_{0\,|\,-1} = S,$$

  constitute the **Kalman filtering algorithm**

- alternatively, (E.39) can be written as

$$\hat{x}_{k\,|\,k} = A_{k-1}\hat{x}_{k-1\,|\,k-1} + \Sigma_{k\,|\,k}C_k'N_k^{-1}(z_k - C_kA_{k-1}\hat{x}_{k-1\,|\,k-1}) \qquad (E.44)$$

- when the system equation contains a control vector u_k

$$x_{k+1} = A_kx_k + B_ku_k + w_k, \quad k = 0, 1, \ldots, N-1$$

  it is straightforward to show that (E.44) takes the form

$$\begin{aligned}\hat{x}_{k\,|\,k} =& A_{k-1}\hat{x}_{k-1\,|\,k-1} + B_{k-1}u_{k-1} \\ &+ \Sigma_{k\,|\,k}C_k'N_k^{-1}(z_k - C_kA_{k-1}\hat{x}_{k-1\,|\,k-1} - C_kB_{k-1}u_{k-1})\end{aligned}$$

  and the equations that (E.41)-(E.43) that generate $\Sigma_{k\,|\,k}$ remain unchanged

186

- finally, we note that (E.41)-(E.42) yield

$$\Sigma_{k+1\,|\,k} = A_k(\Sigma_{k\,|\,k-1} - \Sigma_{k\,|\,k-1}C'_k(C_k\Sigma_{k\,|\,k-1}C'_k + N_k)^{-1}C_k\Sigma_{k\,|\,k-1})A'_k + M_k,$$

with the initial condition $\Sigma_{0\,|\,-1} = S$, which is a matrix Riccati equation

- thus, when A_k, C_k, N_k, and M_k are constant matrices,

$$A_k = A, \quad C_k = C, \quad N_k = N, \quad M_k = M, \quad k = 0, 1, \ldots, N-1,$$

we have that $\Sigma_{k+1\,|\,k}$ tends to a positive definite matrix Σ that solves the algebraic Riccati equation

$$\Sigma = A(\Sigma - \Sigma C'(C\Sigma C' + N)^{-1}C\Sigma)A' + M,$$

assuming observability of the pair (A, C) and controlability of pair (A, D), where $M = DD'$

- under the same conditions, we have $\Sigma_{k\,|\,k} \to \overline{\Sigma}$, where

$$\overline{\Sigma} = \Sigma - \Sigma C'(C\Sigma C' + N)^{-1}C\Sigma$$

and we may write the Kalman filter recursion in the asymptotic form

$$\hat{x}_{k\,|\,k} = A\hat{x}_{k-1\,|\,k-1} + \overline{\Sigma}C'N^{-1}(z_k - CA\hat{x}_{k-1\,|\,k-1})$$

Example 3.4 – Linear Systems and Quadratic Costs

- we will state (without the proofs) that the modified DP algorithm can be used to solve the imperfect state information analog of the linear system/quadratic cost problem of Example 3.1
- we have the same linear system

$$x_{k+1} = A_k x_k + B_k u_k + w_k, \quad k = 0, 1, \ldots, N-1$$

and quadratic cost

$$E\left\{ x_N' Q_N x_N + \sum_{k=0}^{N-1} (x_k' Q_k x_k + u_k' R_k u_k) \right\},$$

but now the controller does not have access to the current state – instead it receives at the beginning of each period k an observation of the form

$$z_k = C_k x_k + v_k, \quad k = 0, 1, \ldots, N-1,$$

where $z_k \in \mathbb{R}^s$, C_k is a given $s \times n$ matrix, and $v_k \in \mathbb{R}^s$ is an observation noise

- furthermore, the vectors v_k are independent, and independent from w_k and x_0 as well, and we make the same assumption about w_k as before – independent, zero mean, finite variance
- the system matrices A_k, B_k are known (otherwise, there is no analytic solution)
- we can write the DP equation (imperfect information) as

$$J_{N-1}(I_{N-1}) = \min_{u_{N-1}} \Big[E_{x_{N-1}, w_{N-1}} \{ x'_{N-1} Q_{N-1} x_{N-1} + u'_{N-1} R_{N-1} u_{N-1}$$

$$(A_{N-1} x_{N-1} + B_{N-1} u_{N-1} + w_{N-1})'$$

$$Q_N (A_{N-1} x_{N-1} + B_{N-1} u_{N-1} + w_{N-1}) \,|\, I_{N-1} \} \Big]$$

- since $E\{w_{N-1} \,|\, I_{N-1}\} = E\{w_{N-1}\} = 0$, it can be written as

$$J_{N-1}(I_{N-1}) = E_{x_{N-1}} \{ x'_{N-1} (A'_{N-1} Q_N A_{N-1} + Q_{N-1}) x_{N-1} \,|\, I_{N-1} \}$$

$$+ E_{w_{N-1}} \{ w'_{N-1} Q_N w_{N-1} \}$$

$$\min_{u_{N-1}} \Big[u'_{N-1} (B'_{N-1} Q_N B_{N-1} + R_{N-1}) u_{N-1}$$

$$+ 2 E\{x_{N-1} \,|\, I_{N-1}\}' A'_{N-1} Q_N B_{N-1} u_{N-1} \Big]$$

- the minimization then yields the optimal policy for the last stage

$$u_{N-1}^* = \mu_{N-1}^*(I_{N-1})$$
$$= -(B_{N-1}'Q_N B_{N-1} + R_{N-1})^{-1} B_{N-1}'Q_N A_{N-1} E\{x_{N-1} \,|\, I_{N-1}\},$$

and upon substitution, we obtain

$$J_{N-1}(I_{N-1}) = E_{x_{N-1}}\left\{ x_{N-1}'K_{N-1}x_{N-1} \,|\, I_{N-1} \right\}$$
$$+ E_{x_{N-1}}\left\{ (x_{N-1} - E\{x_{N-1} \,|\, I_{N-1}\})'P_{N-1}(x_{N-1} - E\{x_{N-1} \,|\, I_{N-1}\}) \,|\, I_{N-1} \right\}$$
$$+ E_{w_{N-1}}\left\{ w_{N-1}'Q_N w_{N-1} \right\},$$

where

$$P_{N-1} = A_{N-1}'Q_N B_{N-1}(R_{N-1} + B_{N-1}'Q_N B_{N-1})^{-1} B_{N-1}'Q_N A_{N-1},$$
$$K_{N-1} = A_{N-1}'Q_N A_{N-1} - P_{N-1} + Q_{N-1}$$

- note that this is almost the same results as we got from the perfect state information, with x_k replaced by its conditional expectation $E\{x_{N-1} \,|\, I_{N-1}\}$ (the middle term corresponds to a penalty for estimation error)

- the same is then done for period $N-2$, and so forth, yielding the optimal policy for every stage:

$$\mu_k^*(I_k) = L_k E\{x_k \mid I_k\},$$

where the matrix L_k is given by
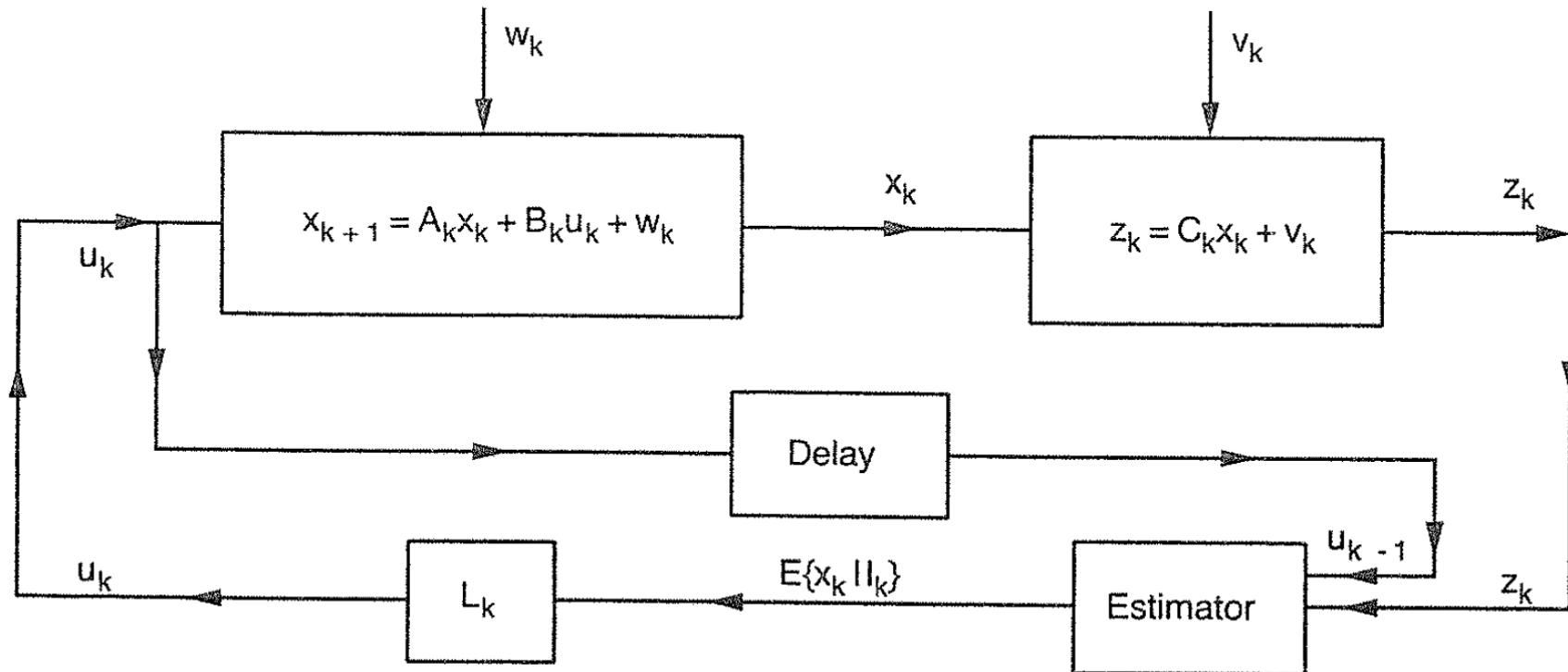
$$L_k = -(R_k + B_k' K_{k+1} B_k)^{-1} B_k' K_{k+1} A_k,$$

and the matrices K_k are given recursively by the Riccati equation

$$K_N = Q_N,$$
$$P_k = A_k' K_{k+1} B_k (R_k + B_k' K_{k+1} B_k)^{-1} B_k' K_{k+1} A_k,$$
$$K_k = A_k' K_{k+1} A_k - P_k + Q_k$$

- again the optimal policy is the same as the one for the perfect information case, except that the state x_k is replaced by its conditional expectation $E\{x_k \mid I_k\}$

- it is interesting to note that the optimal controller can be decomposed into two parts:

a) an **estimator** – uses data to compute $E\{x_k \,|\, I_k\}$

b) an **actuator** – multiplies $E\{x_k \,|\, I_k\}$ by the gain matrix L_k and applies the control input $u_k = L_k E\{x_k \,|\, I_k\}$

- furthermore, the gain matrix L_k is independent of the statistics of the problem and is the same one that would be used if we were faced with the deterministic problem

- on the other hand, as shown in previous section, the estimate $\hat{x}$ of a random vector x given some information I, which minimizes the mean squared error $E_x\{||x - \hat{x}||^2 \,|\, I\}$ is precisely the conditional expectation $E\{x \,|\, I\}$

- thus **the estimator portion of the optimal controller is an optimal solution to the problem of estimating the state x_k assuming no control takes place, while the actuator portion is an optimal solution of the control problem assuming perfect state information prevails**

- this property is called **the separation theorem for linear systems with quadratic cost**

- the linear form of the actuator portion of the controller is particularly attractive for implementation

- in the imperfect information case, however, we need to implement an estimator that produces the conditional expectation

$$\hat{x}_k = E\{x_k \mid I_k\},$$

  which is generally not easy

- in the important special case, where the disturbances w_k, v_k and the initial state x_0 are Gaussian random vectors, a convenient implementation of the estimator is possible by means of the Kalman filter algorithm developed in the previous section

- the algorithm is organized recursively so that to produce $\hat{x}_{k+1}$ at time $k+1$, only the most recent measurement z_{k+1} is needed, together with $\hat{x}_k$ and u_k

- consider finally the case where the system and measurement equations, and the disturbance statistics are stationary, and assume the controllability of (A, B) and (A, C), and observability of (A, F) and (A, D) where $Q = F'F$, $M = DD'$

- then
$$\mu^*(I_k) = L\hat{x}_k,$$

where
$$L = -(R + B'KB)^{-1}B'KA,$$
where K is the unique (PSD) solution of the algebraic Riccati equation
$$K = A'(K - KA(R' + B'KB)^{-1}B'K)A + Q$$

and the steady-state Kalman filtering algorithm:
$$\hat{x}_{k+1} = (A + BL)\hat{x}_k + \overline{\Sigma}C'N^{-1}(z_{k+1} - C(A + BL)\hat{x}_k),$$

where $\overline{\Sigma}$ is given by
$$\overline{\Sigma} = \Sigma - \Sigma C'(C\Sigma C' + N)^{-1}C\Sigma,$$

and Σ is the unique (PSD) solution of the Riccati equation
$$\Sigma = A(\Sigma - \Sigma C'(C\Sigma C' + N)^{-1}C\Sigma)A' + M$$

4 Infinite Horizon Problems

- these problems differ from those considered so far in two aspects:

  (a) the number of stages is infinite

  (b) the system is stationary, i.e., the system equation, the cost per stage, and the random disturbance statistics do not change from one stage to the next

- the assumption of an infinite number of stages is never satisfied in practice, but is a reasonable approximation for problems involving a finite but very large number of stages

- the assumption of stationarity is often satisfied in practice, and in other cases it approximates well a situation where the system parameters vary slowly with time

- infinite horizon problems are interesting because their analysis is elegant and insightful, and the implementation of optimal policies is often simple – for example, optimal policies are typically stationary

- our treatment will be limited to finite-state problems

- there are principal four classes of infinite horizon problems – in the first three, we try to minimize the **total cost over an infinite number of stages**, given by

$$J_\pi(x_0) = \lim_{N \to \infty} E_{w_0, w_1, \ldots} \left\{ \sum_{k=0}^{N-1} \alpha^k g(x_k, \mu_k(x_k), w_k) \right\}$$

- here, $J_\pi(x_0)$ denotes the cost associated with an initial state x_0 and a policy $\pi = \{\mu_0, \mu_1, \ldots\}$, and α is a positive scalar with $0 < \alpha \leq 1$, called the **discount factor**

- the meaning of $\alpha < 1$ is that future costs matter to us less than the same costs incurred at the present time – as an example think of kth period money depreciated to initial period money by a factor of $(1+r)^{-k}$, where r is the rate of interest; here $\alpha = 1/(1+r)$

- an important concern in total cost problems is that the limit in the definition of $J_\pi(x_0)$ be finite

  - in the first two of the following classes of problems, this is guaranteed through various assumptions on the problem structure and the discount factor

  - in the third class, the analysis is adjusted to deal with infinite cost for some of the policies

  - in the fourth class, this sum need not be finite for any policy, and for this reason, the cost is appropriately redefined

(a) **Stochastic shortest path problems**

- here, $\alpha = 1$ but there is a special cost-free termination state; once the system reaches that state it remains there at no further cost

- often, a problem structure such that termination is inevitable is assumed; the horizon is then in effect finite, but its length is random and may be affected by the policy being used

(b) **Discounted problems with bounded cost per stage**

- here, $\alpha < 1$ and the absolute cost per stage $|g(x, u, w)|$ is bounded above by some constant M

- this makes the cost $J_\pi(x_0)$ well defined because it is the infinite sum of a sequence of numbers that are bounded in absolute value by the decreasing geometric progression $\{\alpha^k M\}$

(c) **Discounted and undiscounted problems with unbounded cost per stage**

- here the discount factor α may of may not be less than 1, and the cost per stage may be unbounded

- these problems require a complicated analysis because the possibility of infinite cost for some policies is explicitly dealt with

(d) **Average cost per stage problems**

- minimization of $J_\pi(x_0)$ makes sense only if $J_\pi(x_0)$ is finite for at least some admissible policies π and some initial states x_0

- frequently, however, it turns out that $J_\pi(x_0) = \infty$ for every policy π and initial state x_0 (think of a case where $\alpha = 1$, and the cost for every state and control is positive)

- it turns out that in many such problems the average cost per stage

$$\lim_{N \to \infty} \frac{1}{N} E_{w_0, w_1, \dots} \left\{ \sum_{k=0}^{N-1} g(x_k, \mu_k(x_k), w_k) \right\}$$

is well defined and finite

Preview of Infinite Horizon Results

- there are several analytical and computational issues regarding these problems

- many of these revolve around the relation between the optimal cost-to-go function of the infinite horizon problem and the optimal cost-to-go function of the corresponding N-stage problems

- in particular, consider the case $\alpha = 1$ and let $J_N(x)$ denote the optimal cost of the problem involving N stages, initial state x, cost per stage $g(x, u, w)$, and zero terminal cost – the optimal N-stage cost is generated after N iterations of the DP algorithm

$$J_{k+1}(x) = \min_{u \in U(x)} E_w \left\{ g(x, u, w) + J_k(f(x, u, w)) \right\}, \quad k = 0, 1, \dots,$$

starting from the initial condition $J_0(x) = 0$ for all x (the time indexing notation is reversed to better fit the rest of the theory)

- since the infinite horizon cost of a given policy is the limit of the corresponding N-stage costs as $N \to \infty$, it is natural to speculate that:

(1) the optimal infinite horizon cost converges; that is

$$J^*(x) = \lim_{N \to \infty} J_N(x)$$

for all states x (this holds for problems in categories (a) and (b))

(2) the following limiting form of the DP algorithm holds for all states x:

$$J^*(x) = \min_{u \in U(x)} E_w \left\{ g(x, u, k) + J^*(f(x, u, w)) \right\}$$

Note that this is not really an algorithm, but rather a system of equations, which has as solution the cost-to-go of all the states. It can be also viewed as a **functional equation** for the cost-to-go function J^*, and it is called the **Bellman's equation**

(3) if $\mu(x)$ attains the minimum in the right-hand side of the Bellman's equation for each x, then the policy $\{\mu, \mu, \dots\}$ should be optimal

Total Cost Problem Formulation

- we use the alternative notation from Chapter 1 (p. 12): a controlled finite-state discrete-time dynamic system whereby, at state i, the use of a control u specifies the transition probability $p_{ij}(u)$ to the next state j, and

$$x_{k+1} = w_k,$$

  where w_k is the disturbance ($P\{w_k = j \mid x_k = i, u_k = u\} = p_{ij}(u)$)

- we will suppress w_k from the cost to simplify notation; thus, we assume kth stage cost $g(x_k, u_k)$ for using control u_k at state x_k

- thus, if $\tilde{g}(i, u, j)$ is the cost of using u at state i and moving to state j, we use as cost per stage the expected cost given by

$$g(i, u) = \sum_j p_{ij}(u)\tilde{g}(i, u, j)$$

- the total expected cost associated with an initial state i and a policy $\pi = \{\mu_0, \mu_1, \dots\}$ is

$$J_\pi(i) = \lim_{N \to \infty} E\left\{ \sum_{k=0}^{N-1} \alpha^k g(x_k, \mu_k(x_k)) \mid x_0 = i \right\},$$

  where α is a discount factor with $0 < \alpha \leq 1$

- the optimal cost from state i, the minimum of $J_\pi(i)$ over all admissible policies π, is denoted by $J^*(i)$

- a **stationary policy** is an admissible policy of the form $\pi = \{\mu, \mu, \dots\}$, and its corresponding cost function is denoted by $J_\mu(i)$

- for brevity, we refer to $\{\mu, \mu, \dots\}$ as the stationary policy μ

- we say that μ is optimal if

$$J_\mu(i) = J^*(i) = \min_\pi J_\pi(i), \quad \text{for all states } i$$

Stochastic Shortest Path Problems

- here, we assume no discounting ($\alpha = 1$), and to make the cost meaningful, we assume that **there is a special cost-free termination state** t – once the system reaches that state, it remains there at no further cost, i.e., $p_{tt} = 1$ and $g(t, u) = 0$ for all $u \in U(t)$

- we denote by $1, \ldots, n$ the states other than the termination state t

- we assume that reaching the termination state is inevitable, at least under an optimal policy

- thus, the essence of the problem is to reach the termination state with minimum expected cost

- the deterministic shortest path problem is obtained as the special case where for each state-control pair (i, u), the transition probability $p_{ij}(u)$ is equal to 1 for a unique state j that depends on (i, u)

- certain conditions are required to guarantee that, at least under an optimal policy, termination occurs with probability 1:

> **Assumption:** There exists an integer m such that regardless of the policy used and the initial state, there is positive probability that the termination state will be reached after no more than m stages; that is, for all admissible policies π:
>
> $$\rho_\pi = \max_{i=1,\dots,n} P\{x_m \neq t \mid x_0 = i, \pi\} < 1$$

- the book [p. 406] goes into more detail on how this assumption can be weakened

- under the assumption above, the following results hold for the shortest path problem (proofs are in the book, p. 408-411):

(a) Given any initial conditions $J_0(1), \ldots, J_0(n)$, the sequence $J_k(i)$ generated by the iteration

$$J_{k+1}(i) = \min_{u \in U(i)} \left[g(i, u) + \sum_{j=1}^{n} p_{ij}(u) J_k(j) \right], \quad i = 1, \ldots, n,$$

converges to the optimal cost $J^*(i)$ for each i. [Note that, by reversing the time index this iteration can be viewed as the DP algorithm for a finite horizon problem with terminal cost function equal to J_0.]

(b) The optimal costs $J^*(1), \ldots, J^*(n)$ satisfy Bellman's equation

$$J^*(i) = \min_{u \in U(i)} \left[g(i, u) + \sum_{j=1}^{n} p_{ij}(u) J^*(j) \right], \quad i = 1, \ldots, n,$$

and in fact they are the unique solution of this equation

(c) For any stationary policy μ, the costs $J_\mu(1), \ldots, J_\mu(n)$ are the unique solution of the equation

$$J_\mu(i) = g(i, \mu(i)) + \sum_{j=1}^{n} p_{ij}(\mu(i)) J_\mu(j), \quad i = 1, \ldots, n.$$

Furthermore, given any initial conditions $J_0(1), \ldots, J_0(n)$, the sequence $J_k(i)$ generated by the DP iteration

$$J_{k+1}(i) = g(i, \mu(i)) + \sum_{j=1}^{n} p_{ij}(\mu(i)) J_k(j), \quad i = 1, \ldots, n,$$

converges to the cost $J_\mu(i)$ for each i.

(d) A stationary policy μ is optimal if and only if for every state i, $\mu(i)$ attains the minimum in the Bellman's equation.

Example 4.0 - Spider and a Fly

- a spider and a fly move along a straight line at times $k = 0, 1, \ldots$ with integer initial positions

- at each time period, the fly moves one unit to the left with probability p, one unit to the right with probability p, and stays where it is with probability $1 - 2p$

- the spider knows the position of the fly at the beginning of each period and will always move one unit towards the fly if its distance from the fly is more than one unit

- if the spider is one unit away from the fly, it will either move one unit towards the fly or stay where it is

- if the spider and the fly land in the same position at the end of a period, then the spider captures the fly and the process terminates

- the spider's objective is to capture the fly in minimum expected time

- we view as state the distance between spider and fly - the problem can then be formulated as a stochastic shortest path problem with states $0, 1, \ldots, n$, where n is the initial distance, and a termination state 0, where the spider captures the fly

- denote $p_{1j}(M)$ and $p_{1j}(\overline{M})$ the transition probabilities from state 1 to state j if the spider moves and does not move, and by p_{ij} the transition probabilities from a state $i \geq 2$, then:

$$p_{ii} = p, \quad p_{i(i-1)} = 1 - 2p, \quad p_{i(i-2)} = p, \quad i \geq 2$$

$$p_{11}(M) = 2p, \quad p_{10}(M) = 1 - 2p,$$

$$p_{12}(\overline{M}) = p, \quad p_{11}(\overline{M}) = 1 - 2p, \quad p_{10}(\overline{M}) = p,$$

with all other transition probabilities being 0

- For states $i \geq 2$, Bellman's equation is written as

$$J^*(i) = 1 + pJ^*(i) + (1 - 2p)J^*(i - 1) + pJ^*(i - 2), \quad i \geq 2,$$

where $J^*(0) = 0$ by definition

- the only state where the spider has a choice is when it is one unit away from the fly, and for that state, the Bellman's equation is

$$J^*(1) = 1 + \min[2pJ^*(1), pJ^*(2) + (1 - 2p)J^*(1)],$$

  where the first and second expressions within the bracket are associated with the spider moving and not moving

- by writing the Bellman's equation for $i = 2$, we get

$$J^*(2) = 1 + pJ^*(2) + (1 - 2p)J^*(1),$$

  from which
$$J^*(2) = \frac{1}{1 - p} + \frac{(1 - 2p)J^*(1)}{1 - p}$$

- substituting to the expression above, we get

$$J^*(1) = 1 + \min\left[2pJ^*(1), \frac{p}{1 - p} + \frac{p(1 - 2p)J^*(1)}{1 - p} + (1 - 2p)J^*(1)\right]$$
$$= 1 + \min\left[2pJ^*(1), \frac{p}{1 - p} + \frac{(1 - 2p)J^*(1)}{1 - p}\right]$$

- to solve the preceding equation, we consider two cases where the first expression in the bracket is larger and is smaller than the second expression, i.e.

$$J^*(1) = 1 + 2pJ^*(1),$$

$$2pJ^*(1) \leq \frac{p}{1-p} + \frac{(1-2p)J^*(1)}{1-p},$$

and

$$J^*(1) = 1 + \frac{p}{1-p} + \frac{(1-2p)J^*(1)}{1-p},$$

$$2pJ^*(1) \geq \frac{p}{1-p} + \frac{(1-2p)J^*(1)}{1-p},$$

- the solution to the first case is $J^*(1) = 1/(1-2p)$, and by substituting it to the inequality, we find that this solution is valid when

$$\frac{2p}{1-2p} \leq \frac{p}{1-p} + \frac{1}{1-p},$$

or equivalently (after some calculation), $p \leq 1/3$

- similarly, the solution for the second case is $J^*(1) = 1/p$ and by substituting it to the inequality we find that this solution is valid when

$$2 \geq \frac{p}{1-p} + \frac{1-2p}{p(1-p)},$$

  or equivalently, $p \geq 1/3$

- thus, for $p \leq 1/3$ it is optimal for the spider to move towards the fly and for $p \geq 1/3$ it is optimal to not move (both for situations when the fly is one unit away)

- the minimal expected number of steps for capture when the spider is one unit away from the fly was calculated to be

$$J^*(1) = \begin{cases} 1/(1-2p), & \text{if } p \leq 1/3, \\ 1/p, & \text{if } p \geq 1/3 \end{cases}$$

- given the value of $J^*(1)$, we can calculated the remaining values for $i = 2, \ldots, n$ using the Bellman's equation

Value Iteration for Stochastic Shortest Path Problems

- one of the most frequently used algorithms for solving infinite horizon problems, virtually identical to backward dynamic programming

Step 0. Initialization:

Set $J_0(i) = 0, i = 1, \ldots, n$ [or to any other value].

Fix tolerance parameter $\varepsilon > 0$.

Set $k = 0$.

Step 1. For each $i = 1, \ldots, n$ compute the DP update:
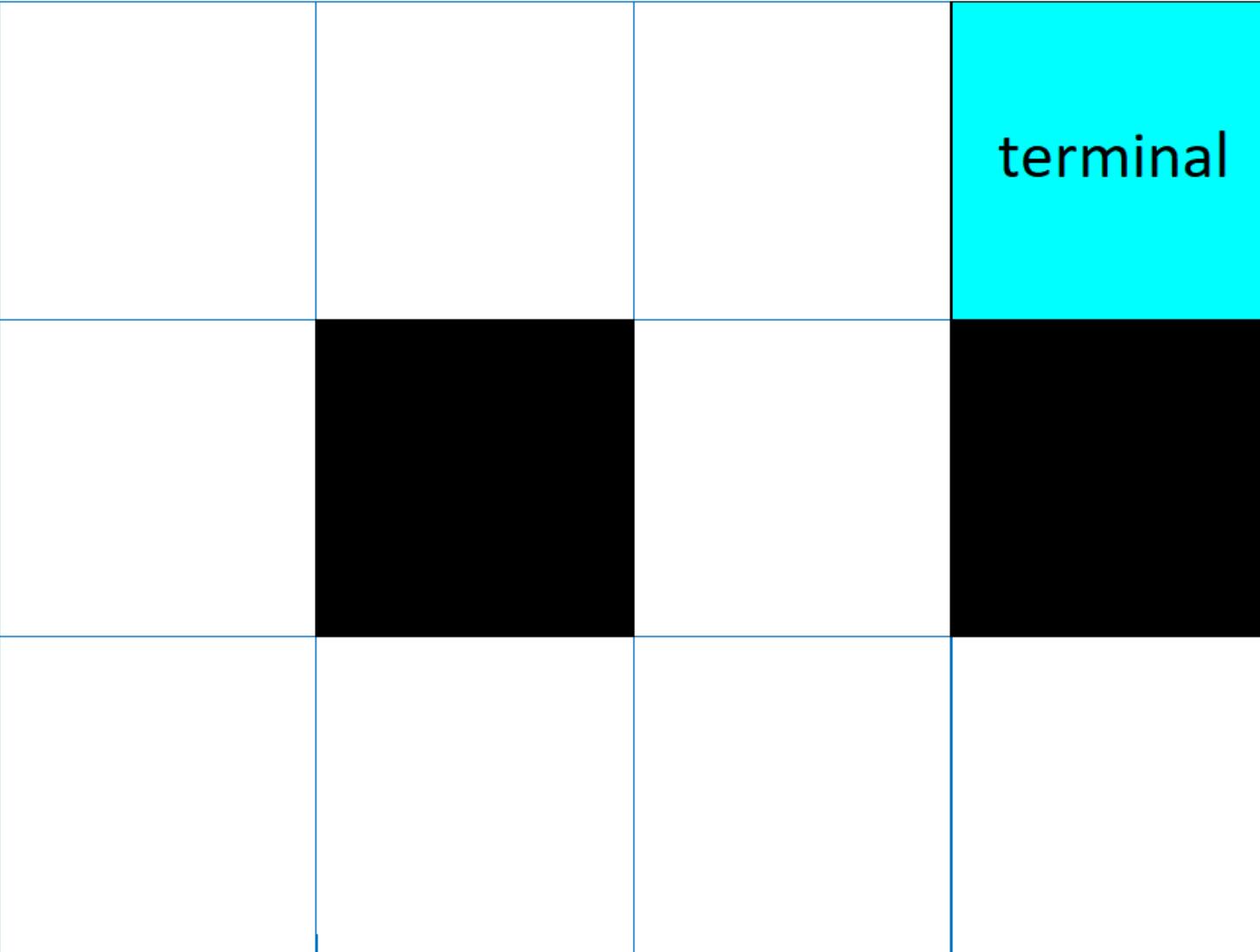
$$J_{k+1}(i) = \min_{u \in U(i)} \left[g(i, u) + \sum_{j=1}^{n} p_{ij}(u) J_k(j) \right]$$

Step 2. Let $\underline{c}_k = \min_{i=1,\ldots,n}[J_{k+1}(i) - J_k(i)], \quad \overline{c}_k = \max_{i=1,\ldots,n}[J_{k+1}(i) - J_k(i)].$

If $||\overline{c}_k - \underline{c}_k|| < \varepsilon$, let μ^ε be the resulting policy and let $J^{*\varepsilon} = J_{k+1}$ and stop; else set $k = k + 1$ and go to Step 1.

Example 4.1 – Gridworld: Shortest Path

- we have an agent (robot, ...) that lives in a grid

- walls and obstacles block its path

- the agent's actions do not always go as planned:

  - 80% of the time, the action "North" takes the agent North (if there is no wall there)
  - 10% of the time, "North" takes the agent West; 10% East
  - if there is a wall (or an obstacle) in the direction the agent would have been taken, the agent stays put
  - other actions work similarly (80% correct, 20% off course)
  - taking a step has a cost equal to 1 (i.e., $g(i, u) = 1$)

- our job is to find an optimal policy that takes the agent from any initial position on the grid to the "terminal" state in the least number of steps

Policy Iteration for Stochastic Shortest Path Problems

- alternative algorithm to value iteration, always terminates finitely

Step 0. Initialization:

Set μ_0 to any (feasible) stationary policy, $k = 0$.

Step 1. Policy evaluation: compute $J_{\mu^k}(i)$ as the solution to the linear system:

$$J(i) = g(i, \mu^k(i)) + \sum_{j=1}^{n} p_{ij}(\mu^k(i)) J(j), \quad i = 1, \ldots, n.$$

Step 2. Policy improvement step:
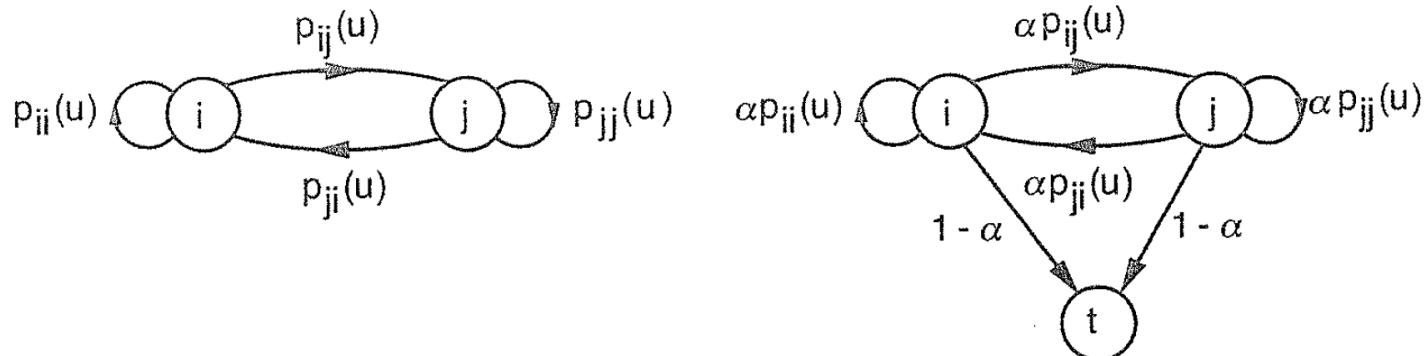
$$\mu^{k+1}(i) = \arg \min_{u \in U(i)} \left[g(i, u) + \sum_{j=1}^{n} p_{ij}(u) J_{\mu^k}(j) \right], \quad i = 1, \ldots, n.$$

Step 3. If $\mu^{k+1} = \mu^k$ (or $J_{\mu^{k+1}} = J_{\mu^k}$) for all states then $\mu^* = \mu^k$; otherwise, set $k = k + 1$ and go to Step 1.

- the linear system of equations of the policy evaluation step can be solved by standard methods such as Gaussian elimination, but when the number of stages is large, this is cumbersome and time-consuming
- a typically more efficient option is to approximate the policy evaluation step with a few value iterations aimed at solving the linear system
- it can be shown that the policy iteration method that uses such approximate policy evaluation yields in the limit the optimal costs and an optimal stationary policy, even if we evaluate each policy using an arbitrary positive number of value iterations
- another possibility for approximating the policy evaluation step is to use simulation - key idea in rollout algorithms discussed in chapter on ADP
- simulations also play a key role in reinforcement learning/neuro-dynamic programming - when the number of states is large, one can try to approximate the cost-to-go function J_{μ^k} by simulating a large number of trajectories under the policy μ^k, and perform some for of least square fit of J_{μ^k} using an approximation architecture

Discounted Problems

- now we consider discounted problems with $\alpha < 1$, and show that this problems can be converted to a stochastic shortest path problem

- let $i = 1, \ldots, n$ be the states and consider an associated stochastic shortest path problem involving the states $1, \ldots, n$ plus an extra termination state t, with state transitions and costs obtained as follows: from a state $i \neq t$, when control u is applied, a cost $g(i, u)$ is incurred, and the next state is j with probability $\alpha p_{ij}(u)$ and t with probability $1 - \alpha$

- suppose that we use the same policy in the discounted problem and in the associated shortest path problem

- then, as long as termination has not occurred, the state evolution in the two problems is governed by the same transition probabilities

- furthermore, the expected cost of the kth stage of the associated shortest path problem is $g(x_k, \mu(x_k))$ multiplied by the probability that state t has not yet been reached, which is α^k (this is also the expected cost of the kth stage for the discounted problem)

- we conclude that the cost of any policy starting from a given state, is the same for the original discounted problem and for the associated stochastic shortest path problem

- this means that we can readily apply the results from the preceding section

a) The value iteration algorithm

$$J_{k+1}(i) = \min_{u \in U(i)} \left[g(i, u) + \alpha \sum_{j=1}^{n} p_{ij}(u) J_k(j) \right], \quad i = 1, \ldots, n,$$

converges to the optimal costs $J^*(i), i = 1, \ldots, n$ starting from arbitrary initial conditions $J_0(1), \ldots, J_0(n)$

b) The optimal costs $J^*(1), \ldots, J^*(n)$ of the discounted problem satisfy Bellman's equation.

$$J^*(i) = \min_{u \in U(i)} \left[g(i, u) + \alpha \sum_{j=1}^{n} p_{ij}(u) J^*(j) \right], \quad i = 1, \ldots, n,$$

and in fact are the unique solution of this equation.

c) For any stationary policy μ, the costs $J_\mu(1), \ldots, J_\mu(n)$ are the unique solution of the equation

$$J_\mu(i) = g(i, u) + \alpha \sum_{j=1}^{n} p_{ij}(\mu(i)) J_\mu(j), \quad i = 1, \ldots, n.$$

Furthermore, given any initial conditions $J_0(1), \ldots, J_0(n)$, the sequence $J_k(i)$ generated by the DP iteration.

$$J_{k+1} = g(i, u) + \alpha \sum_{j=1}^{n} p_{ij}(\mu(i)) J_k(j), \quad i = 1, \ldots, n,$$

converges to the cost $J_\mu(i)$ for each i.

d) A stationary policy μ is optimal if and only if for every state i, $\mu(i)$ attains the minimum in the Bellman's equation.

e) The policy iteration algorithm given by

$$\mu^{k+1}(i) = \arg \min_{u \in U(i)} \left[g(i, u) + \alpha \sum_{j=1}^{n} p_{ij}(u) J_{\mu^k}(j) \right], \quad i = 1 \ldots, n,$$

terminates with an optimal policy.

Example 4.2 – Gridworld: Discounted Problem

- same dynamics as in Example 4.1

- instead of a single terminal state, we consider two terminal states with a reward (-1) in one and a penalty (+1) in the other

- steps do not cost anything (i.e., $g(i, u) = 0$), but we have a discount factor α that makes rewards further in the future less valuable

- our job is to find an optimal policy for each state, minimizing the cost function

Value Iteration – Gauss-Seidel Variant

- a slight variant of the value iteration algorithm that provides a faster rate of convergence

- we take advantage of the fact that when we are computing the expectation of the value of the future, we have to loop over all the states j to compute $\sum_{j=1}^{n} p_{ij}(u) J_k(j)$

- for a particular state i, we would have already computed $J_{k+1}(\hat{i})$ for $\hat{i} = 1, \ldots i - 1$

- by simply replacing $J_k(j)$ with $J_{k+1}(j)$ for the states we have already visited, we obtain an algorithm that typically exhibits a noticeably faster rate of convergence

Average Cost Per Stage Problems

- the methodology of the last two sections applies mainly to problems where the optimal total expected cost is finite either because of discounting or because of a cost-free terminal state that the system eventually enters

- in many situations, however, discounting is inappropriate and there is no natural cost-free state

- in such situations it is often meaningful to optimize the average cost per stage starting from state i, which is defined by

$$J_\pi(i) \lim_{N\to\infty} \frac{1}{N} E\left\{ \sum_{k=0}^{N-1} g(x_k, \mu_k(x_k)) \mid x_0 = i \right\}$$

- for most problems, the average cost per stage of a policy and the optimal average cost per stage are independent of the initial state

a) The optimal average cost λ^* is the same for all initial states and together with some vector $h^* = \{h^*(1), \ldots, h^*(n)\}$ satisfies Bellman's equation

$$\lambda^* + h^*(i) = \min_{u \in U(i)} \left[g(i, u) + \sum_{j=1}^{n} p_{ij}(u) h^*(j) \right], \quad i = 1, \ldots, n.$$

Furthermore, if $\mu(i)$ attains the minimum in the above equation for all i, the stationary policy μ is optimal. In addition, out of all vectors h^* satisfying this equation, there is a unique vector for which $h^*(n) = 0$.

b) If a scalar λ and a vector $h = \{h(1), \ldots, h(n)\}$ satisfy Bellman's equation, then λ is the average optimal cost per stage for each initial state.

c) Given a stationary policy μ with corresponding average cost per stage λ_μ, there is a unique vector $h_\mu = \{h_\mu(1), \ldots, h_\mu(n)\}$ such that $h_\mu(n) = 0$ and

$$\lambda_\mu + h_\mu(i) = g(i, \mu(i)) + \sum_{j=1}^{n} p_{ij}(\mu(i)) h_\mu(j), \quad i = 1, \ldots, n.$$

Value Iteration for Average Cost Per Stage Problems

- a direct application of the value iteration method does not work well for these problems:

  – since typically some of the components of J_k diverge to ∞ or $-\infty$, direct calculation of $\lim_{k\to\infty} J_k(i)/k$ is numerically cumbersome

  – the method will not provide us with a corresponding differential vector h^*

- we can bypass both difficulties by subtracting the same constant from all components of the vector J_k, so that the difference, call it h_k, remains bounded

- in particular, we can consider the algorithm

$$h_k(i) = J_k(i) - J_k(s), \quad i = 1, \ldots, n,$$

  where s is some fixed state

- the modified algorithm is known as **relative value iteration**:

$$h_{k+1}(i) = \min_{u \in U(i)} \left[g(i, u) + \sum_{j=1}^{n} p_{ij}(u) h_k(j) \right] - \min_{u \in U(s)} \left[g(s, u) + \sum_{j=1}^{n} p_{sj}(u) h_k(j) \right]$$

- it can be seen that if the relative value iteration converges to some vector h, then we have

$$\lambda + h(i) = \min_{u \in U(i)} \left[g(i, u) + \sum_{j=1}^{n} p_{ij}(u) h(j) \right],$$

where

$$\lambda = \min_{u \in U(s)} \left[g(s, u) + \sum_{j=1}^{n} p_{sj}(u) h(j) \right],$$

which implies that λ is the optimal cost per stage for all initial states, and h is the associated differential cost vector (convergence is guaranteed under some assumptions, see the book)

229

Policy Iteration for Average Cost Per Stage Problems

- it is possible to use a policy iteration algorithm for the average cost problem and operates similarly to the previous ones: given a stationary policy, we obtain an improved policy by means of a minimization process, and continue until no further improvement is possible

- in the typical step of the algorithm, we have a stationary policy μ_k and perform a policy evaluation step to obtain the corresponding average and differential costs λ^k and $h^k(i)$ satisfying

$$\lambda^k + h^k(i) = g(i, \mu^k(i)) + \sum_{j=1}^{n} p_{ij}(\mu^k(i))h^k(j), \quad i = 1, \ldots, n,$$

$$h^k(n) = 0$$

- we then perform a policy improvement step, finding μ^{k+1} such that

$$g(i, \mu^{k+1}(i)) + \sum_{j=1}^{n} p_{ij}(\mu^{k+1}(i))h^k(j) = \min_{u \in U} \left[g(i, u) + \sum_{j=1}^{n} p_{ij}(u)h^k(j) \right]$$

- if $\lambda^{k+1} = \lambda^k$ and $h^{k+1}(i) = h^k(i)$ for all i, the algorithm terminates

5   Deterministic Continuous-Time Optimal Control

- consider a continuous-time dynamic system

$$\dot{x}(t) = f\left(x(t), u(t)\right), \quad 0 \leq t \leq T, \quad x(0): \text{ given}, \qquad (5.1)$$

  where $x(t) \in \mathbb{R}^n$ is the state vector at time t, $\dot{x}(t) \in \mathbb{R}^n$ is the vector of first order time derivatives of the state at time t, $u(t) \in U \subset \mathbb{R}^m$ is the control vector at time t, U is the control constraint set, and T is the terminal time (all vectors are considered column vectors)

- the components of $f, x, \dot{x}$, and u are denoted as $f_i, x_i, \dot{x}_i$, and u_i

- the system above represents the n first order differential equations

$$\frac{dx_i(t)}{dt} = f_i\left(x(t), u(t)\right), \quad i = 1, \ldots, n$$

- assumptions: system function f_i is continuously differentiable w.r.t. x and continuous w.r.t. u

- the admissible control functions, called **control trajectories**, are the piecewise continuous functions $\{u(t) \mid t \in [0, T]\}$ with $u(t) \in U$ for all $t \in [0, T]$

- additional assumption: the differential equation (5.1) has a unique solution, which is denoted $\{x^u(t) \mid t \in [0, T]\}$ and is called the corresponding **state trajectory** (i.e., we will not care about the existence and uniqueness theorems and all the math behind them)

- the goal is to find an admissible trajectory $\{u(t) \mid t \in [0, T]\}$, which, together with the corresponding state trajectory $\{x(t) \mid t \in [0, T]\}$ minimizes a cost function of the form

$$h\left(x(T)\right) + \int_0^T g\left(x(t), u(t)\right) dt,$$

where g and h are continuously differentiable w.r.t. x, and g is continuous w.r.t. u

Example 5.1 – Unit Mass Movement

A unit mass moves on a line under the influence of a force u. Let $x_1(t)$ and $x_2(t)$ be the position and velocity of the mass at time t. From a given $(x_1(0), x_2(0))$ we want to bring the mass "near" a given final position-velocity pair $(\overline{x}_1, \overline{x}_2)$ at time T. In particular, we want to

$$\text{minimize } |x_1(T) - \overline{x}_1|^2 + |x_2(T) - \overline{x}_2|^2$$

subject to the control constraint

$$|u(t)| \leq 1, \quad \text{for all } t \in [0, T].$$

The corresponding continuous-time system is

$$\dot{x}_1(t) = x_2(t),$$
$$\dot{x}_2(t) = u(t),$$

with the cost functions:

$$h(x(T)) = |x_1(T) - \overline{x}_1|^2 + |x_2(T) - \overline{x}_2|^2,$$
$$g(x(t), u(t)) = 0, \quad \text{for all } t \in [0, T].$$

Example 5.2 – Minimum Time for Unit Mass Movement

The same setting as in the previous example, i.e.

$$\dot{x}_1(t) = x_2(t),$$
$$\dot{x}_2(t) = u(t),$$

with the control constraint

$$|u(t)| \leq 1, \quad \text{for all } t.$$

The minimum time formulation has as the objective the time to reach a certain goal, for instance, stopping at 0, i.e.

$$\text{minimize } T = \int_0^T 1 dt,$$

such that

$$x_1(T) = 0, \quad x_2(T) = 0.$$

- what behavior would you expect an optimal controller to exhibit?

- your intuition might tell you that the best thing that the unit mass can do, to reach the goal in minimum time with limited control input, is to accelerate maximally towards the goal until reaching a critical point, then hit the brakes in order to come to a stop exactly at the goal

- this would be called a **bang-bang control policy**; these are often optimal for systems with bounded input, and it is in fact optimal for this system although we cannot prove it yet

- first, we can figure out the states from which, when the brakes are fully applied, the system comes to rest precisely at the origin

- let's start with the case where $x_1(0) < 0$, and $x_2(0) > 0$, and "hitting the brakes" implies that $u = -1$; integrating the equations, we have

$$\dot{x}_2(t) = u = -1$$
$$\dot{x}_1(t) = x_2(0) - t$$
$$x_1(t) = x_1(0) + x_2(0)t - \frac{1}{2}t^2$$

- substituting $t = x_2(0) - x_2(t)$ into the solution reveals that the system orbits are parabolic arcs:
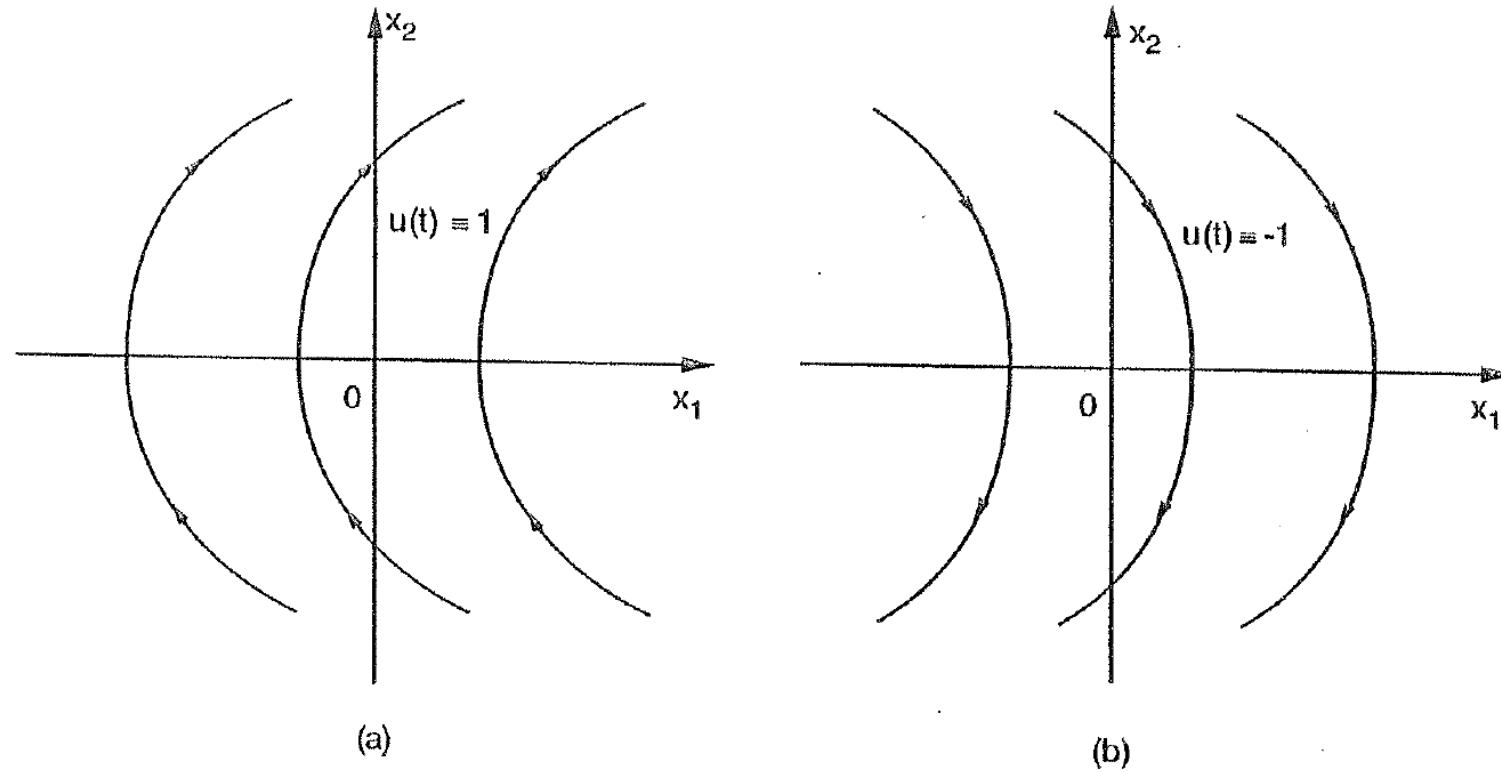
$$x_1 = -\frac{1}{2}x_2^2 + c_-$$

with $c_- = x_1(0) - \frac{1}{2}{x_2}^2(0)$
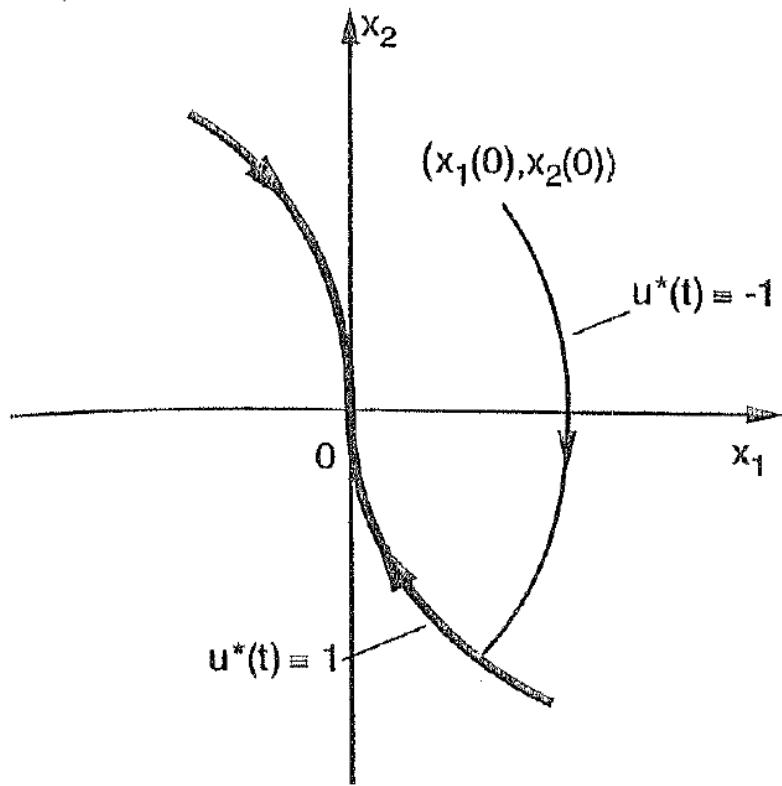
- similarly, the solutions for $u = 1$ are

$$x_1 = \frac{1}{2}x_2 + c_+$$

with $c_+ = x_1(0) + \frac{1}{2}{x_2}^2(0)$

(a)   (b)

- figure above shows state trajectories when control is $u(t) = 1$ (a) and when control is $u(t) = -1$ (b)

- to bring the system from the initial state $(x_1(0), x_2(0))$ to the origin with at most one switch in the value of control, we must apply control according to the following rules involving the switching curve shown on the next slide

a) if the initial state lies above the switching curve, use $u^*(t) = -1$ until the state hits the switching curve, then use $u^*(t) = 1$ until reaching the origin

b) if the initial state lies below the switching curve, use $u^*(t) = 1$ until the state hits the switching curve, then use $u^*(t) = -1$ until reaching the origin
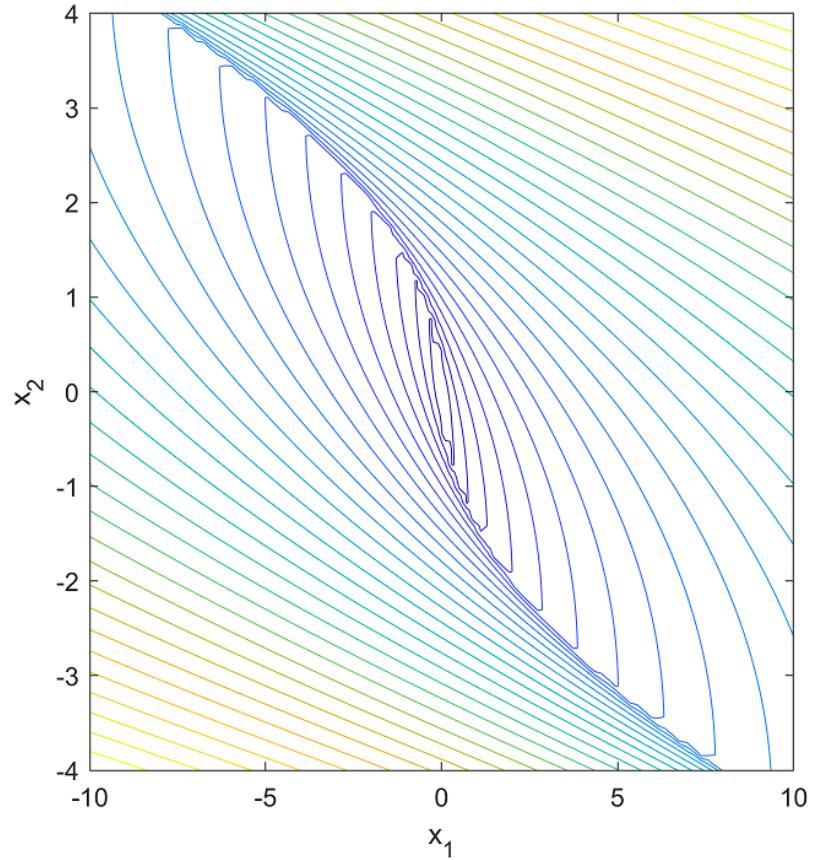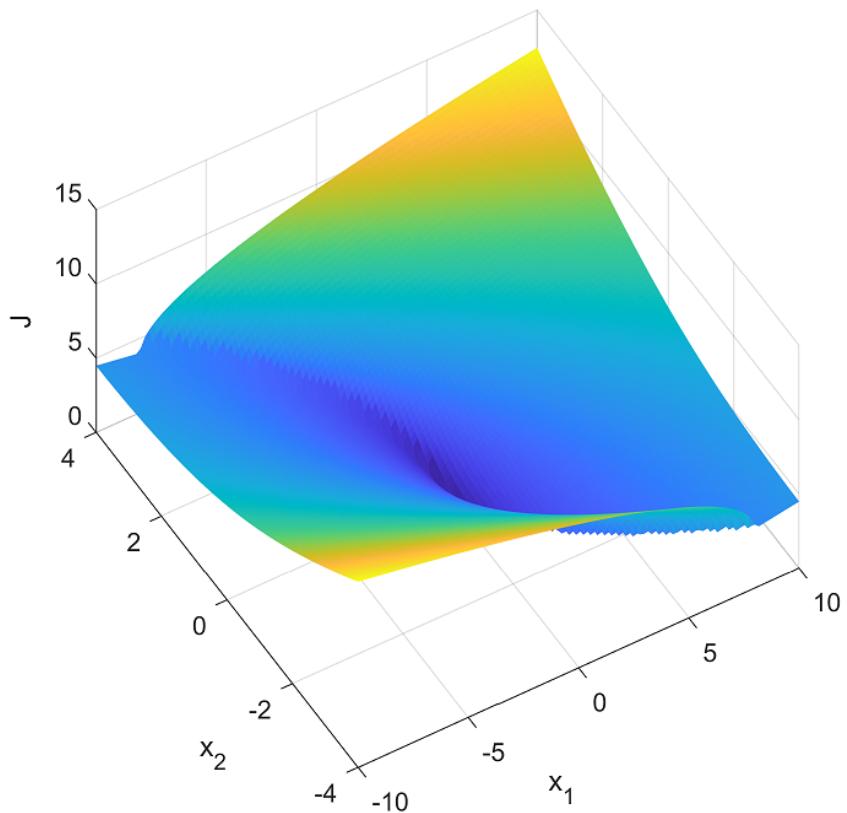
c) if the initial state lies on the top (bottom) part of the switching curve, use $u^*(t) = -1$ $[u^*(t) = 1$, respectively$]$ until reaching the origin

- we can summarize this policy with

$$u(t) = \begin{cases} +1 & \text{if } (x_2(t) < 0 \text{ and } x_1(t) \le \frac{1}{2}x_2(t)^2) \text{ or } (x_2(t) \ge 0 \text{ and } x_1(t) < -\frac{1}{2}x_2(t)^2) \\ 0 & \text{if } x_1(t) = 0 \text{ and } x_2(t) = 0 \\ -1 & \text{otherwise} \end{cases}$$

- we can compute the optimal time to the goal by solving the time required to reach the switching curve plus the time spent moving along the curve to the goal

- with a little algebra, this time to the goal $(J(x))$ is given by

$$J(x) = \begin{cases} 2\sqrt{\frac{1}{2}x_2^2 - x_1} - x_2 & \text{for } u = +1 \text{ regime} \\ 0 & \text{for } u = 0 \\ x_2 + 2\sqrt{\frac{1}{2}x_2^2 + x_1} & \text{for } u = -1 \end{cases}$$

- notice that the function is continuous (though not smooth), even though the policy is discontinuous

The Hamilton-Jacobi-Bellman Equation

- we will derive a continuous-time analog of the DP algorithm

- divide the time horizon $[0, T]$ into N pieces using a discretization interval

$$\delta = \frac{T}{N},$$

denote

$$x_k = x(k\delta), \quad k = 0, 1, \ldots, N,$$
$$u_k = u(k\delta), \quad k = 0, 1, \ldots, N,$$

and approximate continuous-time system by

$$x_{k+1} = x_k + f(x_k, u_k) \cdot \delta$$

and the cost function by

$$h(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k) \cdot \delta$$

241

- now apply DP to the discrete-time approximation: let

$$J^*(t, x) : \text{Optimal cost-to-go at time } t \text{ and state } x$$
$$\text{for the continuous-time problem,}$$
$$\tilde{J}^*(t, x) : \text{Optimal cost-to-go at time } t \text{ and state } x$$
$$\text{for the discrete-time problem,}$$

- the DP equations are
$$\tilde{J}^*(N\delta, x) = h(x),$$
$$\tilde{J}^*(k\delta, x) = \min_{u \in U} \left[g(x, u) \cdot \delta + \tilde{J}^*\big((k+1) \cdot \delta, x + f(x, u) \cdot \delta\big) \right], \quad k = 0, \ldots N - 1$$

- assuming $\tilde{J}^*$ has the required differentiability properties, we expand it into a first order Taylor series around $(k\delta, x)$, obtaining
$$\tilde{J}^*\big((k+1) \cdot \delta, x + f(x, u) \cdot \delta\big) = \tilde{J}^*(k\delta, x) + \nabla_t \tilde{J}^*(k\delta, x) \cdot \delta$$
$$+ \nabla_x \tilde{J}^*(k\delta, x)' f(x, u) \cdot \delta + o(\delta),$$

where $o(\delta)$ are the second order terms with $\lim_{\delta \to 0} o(\delta)/\delta = 0$, ∇_t is the partial derivative w.r.t t, and ∇_x is the n-dimensional partial derivative w.r.t. x

- substituting in the DP, we get

$$\tilde{J}^*(k\delta, x) = \min_{u \in U} \Big[g(x, u) \cdot \delta + \tilde{J}^*(k\delta, x) + \nabla_t \tilde{J}^*(k\delta, x) \cdot \delta$$

$$+ \nabla_x \tilde{J}^*(k\delta, x)' f(x, u) \cdot \delta + o(\delta) \Big]$$

- cancel $\tilde{J}^*(k\delta, x)$ from both sides, divide by δ and take the limit $\delta \to 0$, assuming that the discrete-time cost-to-go function yields in the limit its continuous counterpart,

$$\lim_{k \to \infty, \delta \to 0, k\delta = t} \tilde{J}^*(k\delta, x) = J^*(t, x), \quad \text{for all } t, x,$$

obtain the following equation for the cost-to-go function $J^*(t, x)$:

$$0 = \min_{u \in U} \Big[g(x, u) + \nabla_t J^*(t, x) + \nabla_x J^*(t, x)' f(x, u) \Big], \quad \text{for all } t, x,$$

with the boundary condition $J^*(T, x) = h(x)$

- this is known as the **Hamilton-Jacobi-Bellman (HJB) equation**

- HJB is a **partial differential equation**, which should be satisfied for all time-state pairs (t, x) by the cost-to-go function $J^*(t, x)$, based on the preceding informal derivation (lots of assumptions, e.g. the differentiability of $J^*(t, x)$)

- in fact, we do not know a priori that $J^*(t, x)$ is differentiable, so we do not know if $J^*(t, x)$ solves this equation

- however, it turns out that if we can solve the HJB equation analytically or computationally, then we can obtain an optimal control policy by minimizing its right-hand-side

- this is shown in the following proposition, which is similar to the corresponding statement for discrete DP: if we can execute the DP algorithm (which may not be possible due to excessive computational requirements) we can find the optimal policy by minimizing the right-hand size

Proposition 5.1: (Sufficiency Theorem) Suppose $V(t,x)$ is a solution to the HJB equation; that is, V is continuously differentiable in t and x, and is such that

$$0 = \min_{u \in U} \left[g(x,u) + \nabla_t V(t,x) + \nabla_x V(t,x)' f(x,u) \right], \qquad \text{for all } t, x, \quad (5.2)$$

$$V(T,x) = h(x), \qquad \qquad \qquad \text{for all } x. \quad (5.3)$$

Suppose also that $\mu^*(t,x)$ attains the minimum in (5.2) for all t and x. Let $\{x^*(t) \mid t \in [0,T]\}$ be the state trajectory obtained from the initial condition $x(0)$ when the control trajectory $u^*(t) = \mu^*(t, x(t)), t \in [0,T]$ is used [that is, $x^*(0) = x(0)$ and for all $t \in [0,T], \dot{x}^*(t) = f(x^*(t), \mu^*(t, x^*(t)))$; we assume that this differential equation has a unique solution starting at any pair (t,x) and that the control trajectory $\{\mu^*(t, x^*(t)) \mid t \in [0,T]\}$ is piecewise continuous as a function of t]. Then V is equal to the optimal cost-to-go function, i.e.,

$$V(t,x) = J^*(t,x), \quad \text{for all } x, t.$$

Furthermore, the control trajectory $\{u^*(t) \mid t \in [0,T]\}$ is optimal.

Proof [1/2]: Let $\{\hat{u}(t) \mid t \in [0, T]\}$ be any admissible control trajectory and let $\{\hat{x}(t) \mid t \in [0, T]\}$ be the corresponding state trajectory. From (5.2) we have for all $t \in [0, T]$

$$0 \leq g\big(\hat{x}(t), \hat{u}(t)\big) + \nabla_t V\big(t, \hat{x}(t)\big) + \nabla_x V\big(t, \hat{x}(t)\big)' f\big(\hat{x}(t), \hat{u}(t)\big).$$

Using $\dot{\hat{x}}(t) = f\big(\hat{x}, \hat{u}(t)\big)$, the r.h.s of the above inequality is equal to

$$g\big(\hat{x}(t), \hat{u}(t)\big) + \frac{d}{dt}\big(V(t, \hat{x}(t))\big).$$

where $d/dt(\cdot)$ denotes the total derivative w.r.t. t. Integrating this expression over $t \in [0, T]$ we get

$$0 \leq \int_0^T g\big(\hat{x}(t), \hat{u}(t)\big)\, dt + V(T, \hat{x}(T)) - V(0, \hat{x}(0)).$$

Thus, using the terminal condition $V(T, x) = h(x)$ of (5.3) and the initial condition $\hat{x}(0) = x(0)$, we have

$$V(0, x(0)) \leq h(\hat{x}(T)) + \int_0^T g\big(\hat{x}(t), \hat{u}(t)\big)\, dt.$$

246

Proof [2/2]: If we use $u^*(t)$ and $x^*(t)$ in place of $\hat{u}(t)$ and $\hat{x}(t)$, the inequalities become equalities, and we get

$$V(0, x(0)) = h(x^*(T)) + \int_0^T g\big(x^*(t), u^*(t)\big)\, dt.$$

Therefore, the cost corresponding to $\big\{u^*(t) \,|\, t \in [0, T]\big\}$ is $V(0, x(0))$ and is no larger than the cost corresponding to any other admissible control trajectory $\big\{\hat{u}(t) \,|\, t \in [0, T]\big\}$. It follows that $\big\{u^*(t) \,|\, t \in [0, T]\big\}$ is optimal and that

$$V(0, x(0)) = J^*(0, x(0)).$$

We now note that the preceding argument can be repeated with any initial time $t \in [0, T]$ and any initial state x, We thus obtain

$$V(t, x) = J^*(t, x), \quad \text{for all } t, x.$$

Example 5.2 – Simple Scalar System

The illustrate the HJB equation, consider a simple example of a scalar system

$$\dot{x}(t) = u(t), \quad |u(t)| \leq 1, \text{ for all } t \in [0, T], \quad \text{with cost: } \frac{1}{2}\big(x(T)\big)^2.$$

The HJB equation is

$$0 = \min_{|u| \leq 1} \big[\nabla_t V(t, x) + \nabla_x V(t, x)u\big], \quad \text{for all } t, x, \quad \text{with } V(T, x) = \frac{1}{2}x^2.$$

Candidate solution: move the state towards 0 as quickly as possible, and keep it at 0 once it is at 0. The corresponding control policy:

$$\mu^*(t, x) = -\text{sgn}(x) = \begin{cases} 1, & \text{if } x < 0, \\ 0, & \text{if } x = 0, \\ -1, & \text{if } x > 0. \end{cases}$$

For a given initial time t and initial state x, the cost associated with this policy can be calculated to be

$$J^*(t, x) = \frac{1}{2}\Big(\max\{0, |x| - (T - t)\}\Big)^2,$$

which satisfies the terminal condition, since $J^*(T, x) = (1/2)x^2$.

Now we need to verify that this function satisfies the HJB equation, and that $u = -\text{sgn}(x)$ attains the minimum in the r.h.s. of the equation for all t and x. Proposition 6.1 will then guarantee that the state and control trajectories corresponding to the policy $\mu^*(t, x)$ are optimal. We have

$$\nabla_t J^*(t, x) = \max\{0, |x| - (T - t)\},$$
$$\nabla_x J^*(t, x) = \text{sgn}(x) \cdot \max\{0, |x| - (T - t)\}.$$

After substitution, the HJB becomes

$$0 = \min_{|u| \leq 1} \left[1 + \text{sgn}(x) \cdot u\right] \max\{0, |x| - (T - t)\},$$

which can be seen to hold as an identity for all (t, x). Furthermore, the minimum is attained for $u = -\text{sgn}(x)$. We conclude based on Prop. 6.1 that $J^*(t, x)$ is the optimal cost-to-go function and that the policy μ^* is optimal. Note, however, that the optimal policy is not unique. Based on Prop. 6.1, any policy for which the minimum is attained is optimal. In particular, when $|x(t)| \leq T - t$, applying any control from the range $[-1, 1]$ is optimal.

Example 5.3 – Linear-Quadratic Problems

Consider the n-dimensional linear system

$$\dot{x}(t) = Ax(t) + Bu(t),$$

where A and B are given matrices, and the quadratic cost

$$x(T)'Q_T x(T) + \int_0^T \left(x(t)'Qx(t) + u(t)'Ru(t)\right) dt,$$

where the matrices Q_T and Q are symmetric positive semidefinite, and the matrix R is symmetric positive definite. The HJB equation is

$$0 = \min_{u \in \mathbb{R}^m} \left[x'Qx + u'Ru + \nabla_t V(t,x) + \nabla_x V(t,x)'(Ax + Bu)\right], \qquad (5.4)$$

$$V(T,x) = x'Q_T x. \qquad (5.5)$$

Let us try a solution of the form

$$V(t,x) = x'K(t)x, \quad K(t) : n \times n \text{ symmetric,}$$

and see if we can solve the HJB equation.

We have $\nabla_x V(t, x) = 2K(t)x$ and $\nabla_t V(t, x) = x'\dot{K}(t)x$. Subs into (5.4):

$$0 = \min_u \left[x'Qx + u'Ru + x'\dot{K}(t)x + 2x'K(t)Ax + 2x'K(t)Bu \right]. \qquad (5.6)$$

The minimum is attained at a u for which the gradient w.r.t u is zero:

$$2B'K(t)x + 2Ru = 0, \quad \text{or} \quad u = -R^{-1}B'K(t)x. \qquad (5.7)$$

Substitute the minimizing value into (5.6), we get

$$0 = x'\left(\dot{K}(t) + K(t)A + A'K(t) - K(t)BR^{-1}B'K(t) + Q \right)x, \text{ for all } (t, x).$$

In order for $V(t, x) = x'K(t)x$ to solve the HJB equation, $K(t)$ must satisfy a matrix differential equation, known as the **continuous-time Riccati equation**

$$\dot{K}(t) = -K(t)A - A'K(t) + K(t)BR^{-1}B'K(t) - Q, \quad K(T) = Q_T. \qquad (5.8)$$

Reverse: if $K(t)$ is a solution to (5.8), then $V(t, x) = x'K(t)x$ is a solution to the HJB equation. Thus, by Prop. 3.2.1, the optimal cost-to-go function and optimal policy are

$$J^*(x, t) = x'K(t)x, \quad \mu^*(t, x) = -R^{-1}B'K(t)x.$$

The Pontryagin Minimum Principle

- we use an informal derivation using the HJB equation (derivation based on variational methods is in the book)

$$0 = \min_{u \in U} \left[g(x, u) + \nabla_t J^*(t, x) + \nabla_x J^*(t, x)' f(x, u) \right], \quad \text{for all } t, x, \quad (5.9)$$

$$J^*(T, x) = h(x), \quad \text{for all } x \quad (5.10)$$

- we argued that the optimal cost-to-go function $J^*(t, x)$ satisfies this equation under some conditions and the sufficiency theorem suggests that if for a given initial state $x(0)$, the control trajectory $\{u^*(t) \mid t \in [0, T]\}$ is optimal with the corresponding state trajectory $\{x^*(t) \mid t \in [0, T]\}$, then for all $t \in [0, T]$

$$u^*(t) = \arg \min_{u \in U} \left[g(x^*(t), u) + \nabla_x J^*(t, x^*(t))' f(x^*(t), u) \right] \quad (5.11)$$

- note that to obtain the optimal control trajectory via this equation, we do not need to know $\nabla_x J^*$ at **all** values of x and t; it is sufficient to know $\nabla_x J^*$ at only **one** value of x for each t, that is, to know only $\nabla_x J^*(t, x^*(t))$

- the Minimum Principle is basically the equation (5.11)

- it turns out we can often calculate $\nabla_x J^*(t, x^*(t))$ along the optimal state trajectory far more easily than we can solve the HJB equation

- in particular, $\nabla_x J^*(t, x^*(t))$ satisfies a certain differential equation, called the **adjoint equation**

- we derive this equation by differentiating (5.9), but first we need to know how to differentiate functions involving minima (proof in the book, p. 116)

Lemma 5.2: Let $F(t, x, u)$ be a continuously differentiable function of $t \in \mathbb{R}$, $x \in \mathbb{R}^n$, and $u \in \mathbb{R}^m$, and let U be a convex subset of $\mathbb{R}^m$. Assume that $\mu^*(t, x)$ is a continuously differentiable function such that

$$\mu^*(t, x) = \arg\min_{u \in U} F(t, x, u), \quad \text{for all } t, x.$$

Then

$$\nabla_t \left\{ \min_{u \in U} F(t, x, u) \right\} = \nabla_t F\big(t, x, \mu^*(t, x)\big), \quad \text{for all } t, x,$$

$$\nabla_x \left\{ \min_{u \in U} F(t, x, u) \right\} = \nabla_x F\big(t, x, \mu^*(t, x)\big), \quad \text{for all } t, x.$$

[Note: On the l.h.s., $\nabla_t\{\cdot\}$ and $\nabla_x\{\cdot\}$ denote the gradients of the function $G(t, x) = \min_{u \in U} F(t, x, u)$ w.r.t. t and x. On the r.h.s.m ∇_t and ∇_x denote the vectors of partial derivatives of F w.r.t. t and x, evaluated at $(t, x, \mu^*(t, x))$.]

254

- consider the HJB equation (5.9), and for any (t, x), suppose that $\mu^*(t, x)$ is the optimal control

- restrictive assumptions: U is a convex set, and $\mu^*(t, x)$ is continuously differentiable in (t, x) (this is not needed for the variational derivation)

- differentiate both sides of the HJB equation w.r.t x and t; in particular, set to zero the gradient w.r.t. x and t of the function

$$g(x, \mu^*(x, t)) + \nabla_t J^*(t, x) + \nabla_x J^*(t, x)' f(x, \mu^*(x, t)),$$

and rely on Lemma 6.2 to disregard the terms involving the derivatives of $\mu^*(t, x)$ w.r.t. t and x

- obtain

$$0 = \nabla_x g(x, \mu^*(t, x)) + \nabla_{xt}^2 J^*(t, x) + \nabla_{xx}^2 J^*(t, x) f(x, \mu^*(t, x))$$

$$+ \nabla_x f(x, \mu^*(t, x)) \nabla_x J^*(t, x) \tag{5.12}$$

$$0 = \nabla_{tt}^2 J^*(t, x) + \nabla_{xt}^2 J^*(t, x)' f(x, \mu^*(t, x)), \tag{5.13}$$

where $\nabla_x f(x, \mu^*(t, x))$ is the matrix

$$\nabla_x f = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_1} \\ \vdots & \vdots & \vdots \\ \frac{\partial f_1}{\partial x_n} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

with the partial derivatives evaluated at $(x, \mu^*(t, x))$

- these equations hold for all (t, x), but we are going to focus only on the optimal state and control trajectory

- the optimal state and control trajectory $\{(x^*(t), u^*(t)) \,|\, t \in [0, T]\}$, where $u^*(t) = \mu^*(t, x^*(t))$ for all $t \in [0, T]$

- we have for all t
$$\dot{x}^*(t) = f(x^*(t), u^*(t)),$$
so that the term
$$\nabla^2_{xt} J^*(t, x^*(t)) + \nabla^2_{xx} J^*(t, x^*(t)) f(x^*(t), u^*(t))$$
in (5.12) is equal to the following total derivative w.r.t. t
$$\frac{d}{dt}\left(\nabla_x J^*(t, x^*(t))\right)$$

- similarly, the term
$$\nabla^2_{tt} J^*(t, x^*(t)) + \nabla^2_{xt} J^*(t, x^*(t))' f(x^*(t), u^*(t))$$
in (5.13) is equal to the total derivative
$$\frac{d}{dt}\left(\nabla_t J^*(t, x^*(t))\right)$$

- denote

$$p(t) = \nabla_x J^*(t, x^*(t)), \tag{5.14}$$

$$p_0(t) = \nabla_t J^*(t, x^*(t)), \tag{5.15}$$

then (5.12) becomes

$$\dot{p}(t) = -\nabla_x f(x^*(t), u^*(t))p(t) - \nabla_x g(x^*(t), u^*(t)) \tag{5.16}$$

and (5.13) becomes

$$\dot{p}_0(t) = 0, \tag{5.17}$$

or equivalently: $p_0(t) = \text{constant}$, for all $t \in [0, T]$

- Eq (5.16) is a system of n first order differential equations known as the **adjoint equation**

- from the boundary condition

$$J^*(T, x) = h(x), \quad \text{for all } x,$$

we get, by differentiating w.r.t. x the relation $\nabla_x J^*(T, x) = \nabla h(x)$, and by using $\nabla_x J^*(t, x^*(t)) = p(t)$, we get

$$p(T) = \nabla h(x^*(T)), \tag{5.18}$$

which is a boundary condition for the adjoint equation (5.16)

- summary: along optimal state and control trajectories $x^*(t)$, $u^*(t)$, $t \in [0, T]$, the adjoint equation (5.16) holds together with the boundary condition (5.18), while (5.11) and the definition of $p(t)$ imply that $u^*(t)$ satisfies

$$u^*(t) = \arg\min_{u \in U} \left[g(x^*(t), u) + p(t)' f(x^*(t), u) \right], \quad \text{for all } t \in [0, T] \tag{5.19}$$

Hamiltonian formulation

- motivated by the condition (5.19), introduce the Hamiltonian function mapping triplets $(x, u, p) \in \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n$ to real numbers and given by

$$H(x, u, p) = g(x, u) + p' f(x, u)$$

- note that both the system and the adjoint equations can be compactly written in terms of the Hamiltonian as

$$\dot{x}^*(t) = \nabla_p H(x^*(t), u^*(t), p(t)),$$
$$\dot{p}(t) = -\nabla_x H(x^*(t), u^*(t), p(t))$$

- the Minimum Principle is stated in terms of the Hamiltonian function

Proposition 5.3 (Minimum Principle): Let $\{u^*(t) \mid t \in [0, T]\}$ be an optimal control trajectory and let $\{x^*(t) \mid t \in [0, T]\}$ be the corresponding state trajectory, i.e.

$$\dot{x}^*(t) = f(x^*(t), u^*(t)), \quad x^*(0) = x(0) \; : \; \text{given}.$$

Let also $p(t)$ be a solution to the adjoint equation

$$\dot{p}(t) = -\nabla_x H(x^*(t), u^*(t), p(t)),$$

with the boundary condition

$$p(T) = \nabla h(x^*(T)),$$

where $h(\cdot)$ is the terminal cost function. Then, for all $t \in [0, T]$,

$$u^*(t) = \arg \min_{u \in U} H(x^*(t), u, p(t)).$$

Furthermore, there is a constant C such that

$$H(x^*(t), u^*(t), p(t)) = C, \quad \text{for all } t \in [0, T].$$

- to see why the Hamiltonian function is constant for $t = [0, T]$ along the optimal state and control trajectories, note that by Eqs. (5.9), (5.14), and (5.15), we have for all $t \in [0, T]$

$$H(x^*(t), u^*(t), p(t)) = -\nabla_t J^*(t, x^*(t)) = -p_0(t),$$

  and $p_0(t)$ is constant by Eq. (5.17)

- note that the Hamiltonia function need not be constant along the optimal trajectory if the system and cost are not time-independent

- the Minimum Principle provides **necessary** optimality conditions – all optimal trajectories must satisfy these conditions, but if a trajectory satisfies the conditions, it is not necessarily optimal (further analysis is needed)

- when the system function f is linear in (x, u), the constraint set U is convex, and the cost functions h and g are convex – the conditions are both sufficient and necessary

- the Minimum Principle can often be used as the basis of a numerical solution

- one possibility is the **two-point boundary problem method**: we use the minimum condition

$$u^*(t) = \arg \min_{u \in U} H(x^*(t), u, p(t)),$$

to express $u^*(t)$ in terms of $x^*(t)$ and $p(t)$, and then substitute the result into the system and the adjoint equations, to obtain a set of $2n$ first order differential equations in the components $x^*(t)$ and $p(t)$

- these equations can be solved using the split boundary conditions

$$x^*(0) = x(0), \quad p(T) = \nabla h(x^*(T))$$

- which could be solved numerically (although with a bit of difficulty)

- using the Minimum Principle to obtain analytical solutions is possible in some situations, but requires intense creativity and skill

Example 5.4 – LQP revisited

Consider the one-dimensional linear system

$$\dot{x}(t) = ax(t) + bu(t),$$

where a and b are given scalars. Find an optimal control over $[0, T]$ that minimizes the quadratic cost

$$\frac{1}{2}q\big(x(T)\big)^2 + \frac{1}{2}\int_0^T \big(u(t)\big)^2 dt,$$

where q is a given positive scalar. There are no constraints on the control, so we have a special case of the Example 6.3. We will solve this problem via the Minimum Principle. The Hamiltonian and the adjoint equation are

$$H(x, u, p) = \frac{1}{2}u^2 + p(ax + bu),$$

$$\dot{p}(t) = -ap(t), \quad p(T) = qx^*(T)$$

The optimal control is obtained by minimizing the Hamiltonian

$$u^*(t) = \arg\min \left[\frac{1}{2} u^2 + p(t)(ax^*(t) + bu) \right] = -bp(t) \qquad (5.20)$$

We will get the optimal solution from these conditions using two different approaches. The first one will use the two-point boundary problem: We eliminate the control from the system equation to get

$$\dot{x}^*(t) = ax^*(t) - b^2 p(t)$$

From the adjoint equation we get

$$p(t) = e^{-at}\xi, \quad \text{for all } t \in [0, T],$$

where $\xi = p(0)$ is an unknown parameter. Combining the last tow equations we get

$$\dot{x}^*(t) = ax^*(t) - b^2 e^{-at}\xi, \quad x^*(0) = x(0), \quad x^*(T) = \frac{e^{-aT}\xi}{q}. \qquad (5.21)$$

It can be verified that the solution of (5.21) is given by

$$x^*(t) = x(0)e^{at} + \frac{b^2\xi}{2a}(e^{-at} - e^{at}),$$

and ξ can be obtained from the last two relations:

$$x^*(T) = x(0)e^{aT} + \frac{b^2\xi}{2a}(e^{-aT} - e^{aT}) = \frac{e^{-aT}\xi}{q}$$

$$\xi = \frac{x(0)e^{aT}}{\frac{e^{-aT}}{q} - \frac{b^2}{2a}(e^{-aT} - e^{aT})}$$

Given ξ, we get $p(t) = e^{-at}\xi$, and from $p(t)$, we can determine the optimal control trajectory from (5.20) as $u^*(t) = -bp(t) = -be^{-at}\xi$.

In the second approach, we derive the Riccati equation from the previous example. In particular, we hypothesize a linear relation between $x^*(t)$ and $p(t)$

$$K(t)x^*(t) = p(t), \quad \text{for all } t \in [0, T], \tag{5.22}$$

and we show that $K(t)$ can be obtained by solving Riccati equation. Indeed, from (5.20) we have

$$u^*(t) = -bK(t)x^*(t),$$

which by substitution yields

$$\dot{x}^*(t) = (a - b^2 K(t))x^*(t).$$

By differentiating (5.22) and by using the adjoint equation , we get

$$\dot{K}(t)x^*(t) + K(t)\dot{x}^*(t) = \dot{p}(t) = -ap(t) = -aK(t)x^*(t)$$

By combining the last two relations, we get

$$\dot{K}(t)x^*(t) + K(t)(a - b^2 K(t))x^*(t) = -aK(t)x^*(t),$$

from which we see that $K(t)$ should satisfy

$$\dot{K}(t) = -2aK(t) + b^2(K(t))^2, \quad K(T) = q.$$

This is the Ricatti equation of the previous example. It can be solved analytically (see Bernoulli differential equation) – substitute $v = K^{-1}$, then

$$\dot{v} = -\frac{\dot{K}}{K^2} = \frac{2a}{K} - b^2 = 2av - b^2,$$

which is a "simple" linear ODE. Once $K(t)$ is known, the optimal control is obtained in the closed-loop form

$$u^*(t) = -bK(t)x^*(t).$$

By reversing the preceding arguments, this control can then be shown to satisfy all the conditions of the Minimum Principle.

Example 5.5 – Minimum Time Revisited

- we now have the machinery that can be used to verify the bang-bang policy for the minimum time problem we saw in Example 5.2, i.e.

$$\dot{x}_1(t) = x_2(t),$$
$$\dot{x}_2(t) = u(t),$$
$$|u(t)| \leq 1, \quad \text{for all } t$$

with the objective

$$\text{minimize } T = \int_0^T 1 dt \quad (\text{i.e. } g(x(t), u(t)) = 1)$$

- the initial state $(x_1(0), x_2(0))$ is given and the terminal state is also given

$$x_1(T) = 0, \quad x_2(T) = 0$$

- if $\{u^*(t) \mid t \in [0, T]\}$ is an optimal control trajectory, $u^*(t)$ must minimize the Hamiltonian for each t, i.e.

$$u^*(t) = \arg \min_{-1 \le u \le 1} [1 + p_1(x)x_2^*(t) + p_2(t)u]$$

  therefore

$$u^*(t) = \begin{cases} 1 & \text{if } p_2 < 0, \\ -1 & \text{if } p_2 \ge 0. \end{cases}$$

- the adjoint equation is
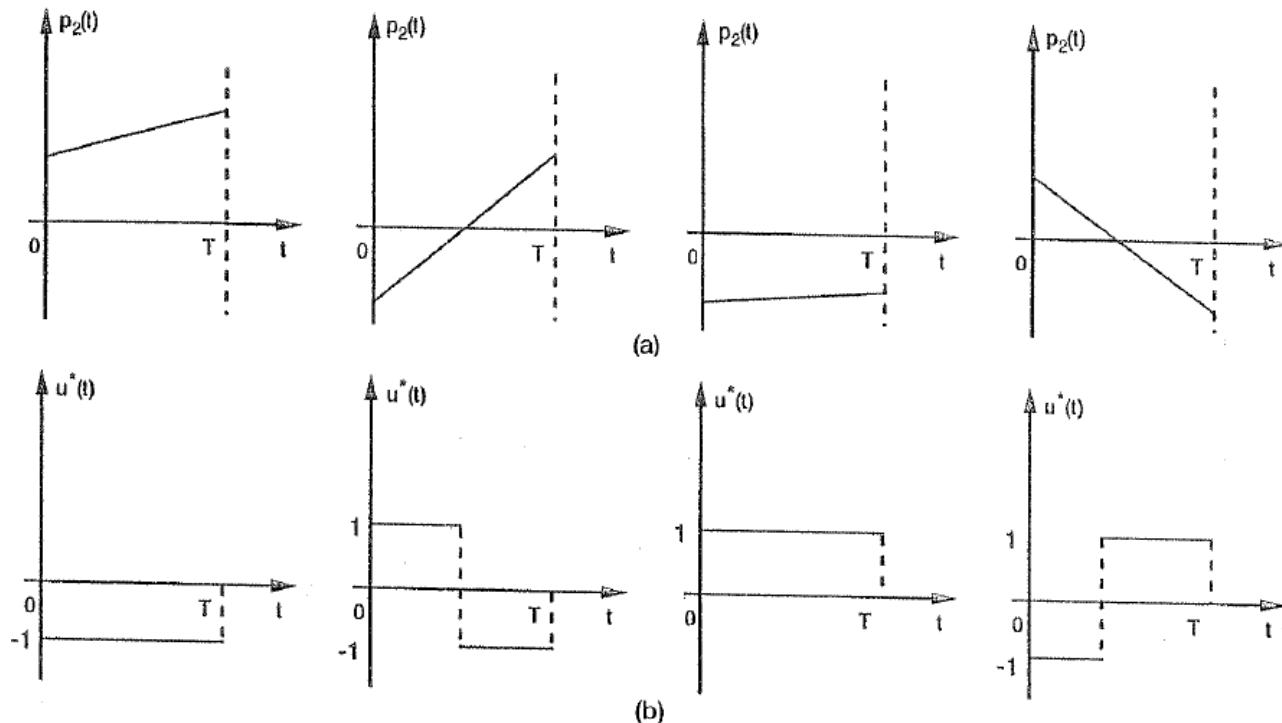
$$\dot{p}_1(t) = 0, \quad \dot{p}_2(t) = -p_1(t),$$

  so

$$p_1(t) = c_1, \quad p_2(t) = c_2 - c_1 t,$$

  where c_1 and c_2 are constants

- it follows that $\{p_2(t) \mid t \in [0, T]\}$ has one of the forms shown on the figure on the next slide; that is, it switches at most once in going from negative to positive and reversely [note that it is not possible for $p_2(t)$ to be equal to zero for all t, as that would imply that $p_1(t)$ is also equal to zero, so that the Hamiltonian would be equal to 1]

(a)

(b)

- the figure above shows (a) possible forms of the adjoint variable $p_2(t)$, (b) corresponding forms of the optimal control trajectory

- for the precise form of the optimal control trajectory, we use the given initial and final states: for $u(t) = \zeta$, where $\zeta = \pm 1$, the system evolves according to

$$x_1(t) = x_1(0) + x_2(0)t + \frac{\zeta}{2}t^2, \quad x_2(t) = x_2(0) + \zeta t$$

271

6   Metaheuristics

- we have seen the use of DP for a certain kind of well-structured problems

- we have also seen that, in some situations, it might be more efficient to use a Branch-and-Bound method or various shortest path algorithms (discussed in Chapter 2) when a particular problem structure can be exploited

- in a similar fashion, there are problems where neither DP nor any of the other approaches for well-structured problems can be readily used

- in this chapter, we will look at **metaheuristics** - a class of methods with relatively simple structure that has found extreme success in such cases

- our treatment will closely follow a book "Essentials of Metaheuristics" by S. Luke[1]

---

[1] Available online at `https://cs.gmu.edu/~sean/book/metaheuristics/Essentials.pdf`
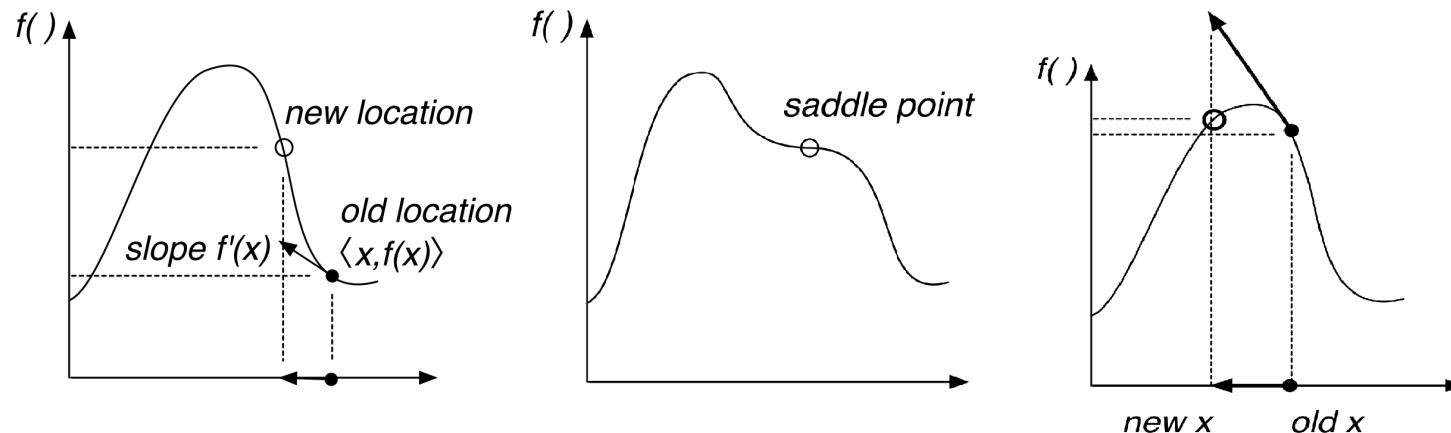
Gradient-Based Optimization

- we start by refreshing the simple concepts from the traditional mathematical method for finding the maximum (the whole book is about maximization, for the purpose of not having to redo the figures, we will stick with it) of a function

- possibly the simplest algorithm is Gradient Ascent, where the idea is to identify the slope of the function we want to maximize ($f(x)$) and move up it

- this method doesn't require us to compute or even know $f(x)$, but it does assume we can compute the slope of x, that is, we have $\nabla f(x)$.

- the technique is rather simple - we start with an arbitrary value for x and we then repeatedly add to it a small portion of its slope, that is, $x \leftarrow x + \alpha \nabla f(x)$, where α is a very small positive value (possibly found by a 1D search routine)

273

Algorithm 1 Gradient Ascent

$x \leftarrow$ random initial vector
repeat
   $x \leftarrow x + \alpha \nabla f(x)$
until x is the ideal solution or we have run out of time
return x

- note that the algorithm runs until we've found "the ideal solution" or "we have run out of time"

- there are several problems with this approach - "0" slope might mean local maximum, minimum or saddle point, the algorithm can sometimes "overshoot", etc

- how can we escape local optima?

- with the tools of gradient ascent, there's really only one way: change α to a sufficiently large value that the algorithm potentially overshoots not only the top of its hill but actually lands on the "next hill"

- alternatively, we could put Gradient Ascent in a big loop: each time we start with a random starting point, and end when we've reached a local optimum

- we keep trying over and over again, and eventually return the best solution discovered

- to determine what the "best solution discovered" is, we need to be able to compute $f(x)$ (which was not required before), so we can compare results

- this little modification now results in a **global optimization algorithm** - guaranteed to find the global optimum if it runs long enough

Algorithm 2 Gradient Ascent with Restarts
   $x \leftarrow$ random initial vector
   $x^* \leftarrow x$
   repeat
      repeat
      $x \leftarrow x + \alpha \nabla f(x)$
      until $||\nabla f(x)|| = 0$ (or $<$ tolerance)
      if $f(x) > f(x^*)$ **then**
         $x^* \leftarrow x$
      $x \leftarrow$ random value
   until we have run out of time
   return x^*

- at the limit, at some point, Gradient Ascent with Restarts will discover the optimum because, at the very least, some restart will randomly land right on the optimum, just like random search

- more realistically, a random restart will eventually start on the globally optimal hill, allowing gradient ascent to climb to the optimum

Single-State Methods

- gradient-based optimization makes a big assumption: that we can compute the first derivative, which is reasonable when optimizing a well-formed, well-understood mathematical function

- but in many cases, we can't compute the gradient of the function because we don't even know what the function is - a so-called "**black-box**" setting

- to optimize a candidate solution in this scenario, we need to be able to:

  - provide one or more initial candidate solutions (**initialization**)
  - assess the **quality** of a candidate solution (**assessment**)
  - make a **copy** of a candidate solution
  - tweak a candidate solution, which produces a randomly slightly different candidate solution (**modification**)

- to this the algorithm will typically provide a **selection** procedure that decides which candidate solutions to retain and which to reject

277

Hill-Climbing

- this technique is related to gradient ascent, but it doesn't require us to know the strength of the gradient or even its direction: we just iteratively test new candidate solutions in the region of your current candidate, and adopt the new ones if they're better

- this enables us to climb up the hill until we reach a local optimum

- notice the strong resemblance between Hill-Climbing and Gradient Ascent - the only difference is that Hill-Climbing's more general Tweak operation must instead rely on a stochastic (partially random) approach to hunting around for better candidate solutions

---
Algorithm 3 Hill-Climbing

$S \leftarrow$ initial candidate solution   ▷ **initialization**
repeat
   $R \leftarrow \text{Tweak}(\text{Copy}(S))$ ▷ **modification**
   if $\text{Quality}(R) > \text{Quality}(S)$ **then**   ▷ **assessment and selection**
      $S \leftarrow R$
until S is the ideal solution or we have run out of time
return S

- ee can make this algorithm a little more aggressive: create n "tweaks" to a candidate solution all at one time, and then possibly adopt the best one

- this modified algorithm is called **Steepest Ascent Hill-Climbing**, because by sampling all around the original candidate solution and then picking the best, we're essentially sampling the gradient and marching straight up it

---

Algorithm 4 Steepest Ascent Hill-Climbing

$n \leftarrow$ number of tweaks desired to sample the gradient

$S \leftarrow$ initial candidate solution
repeat
   $R \leftarrow \text{Tweak}(\text{Copy}(S))$
   for $n - 1$ times **do**
      $W \leftarrow \text{Tweak}(\text{Copy}(S))$
      if $\text{Quality}(W) > \text{Quality}(R)$ **then**
         $R \leftarrow W$
   if $\text{Quality}(R) > \text{Quality}(S)$ **then**
      $S \leftarrow R$
until S is the ideal solution or we have run out of time
return S

- a popular variation, called **Steepest Ascent Hill-Climbing with Replacement** (in the book), is to not bother comparing R to S: instead, we just replace S directly with R

- this runs the risk of losing our best solution of the run, so we'll augment the algorithm to keep the best-discovered-so-far solution stashed away, in a reserve variable called $Best$ (which we return at the end)

Algorithm 5 Steepest Ascent Hill-Climbing with Replacement

$n \leftarrow$ number of tweaks desired to sample the gradient

$S \leftarrow$ initial candidate solution
$Best \leftarrow S$
repeat
  $R \leftarrow$ Tweak(Copy(S))
  for $n - 1$ times **do**
    $W \leftarrow$ Tweak(Copy(S))
    if Quality(W) > Quality(R) **then**
      $R \leftarrow W$
  $S \leftarrow R$
  if Quality(S) > Quality($Best$) **then**
    $Best \leftarrow S$
until $Best$ is the ideal solution or we have run out of time
return $Best$

- the initialization, Copy, Tweak, and (to a lesser extent) fitness assessment functions collectively define the **representation** of our candidate solution

- together they stipulate what our candidate solution is made up of and how it operates

- what might a candidate solution look like?

  - a vector

  - an arbitrary-length list of objects

  - an unordered set or collection of objects

  - a tree

  - a graph

- or any combination of the above (whatever seems to be appropriate to the problem we are trying to solve)

- to Tweak a vector of real numbers we might (as one of many possibilities) add a small amount of uniform random noise with range $[-r, r]$ to each number

- r now becomes "a knob" we can turn

  - if the size is very small, then Hill-Climbing will march right up a local hill and be unable to make the jump to the next hill because the bound is too small for it to jump that far

  - on the other hand, if the size is large, then Hill-Climbing will bounce around a lot, and when it is near the top of a hill, it will have a difficult time converging to the peak, as most of its moves will be so large as to overshoot the peak

- thus small sizes of the bound move slowly and get caught in local optima; and large sizes on the bound bounce around too frenetically and cannot converge rapidly to finesse the very top of peaks

- this knob is one way of controlling the degree of **Exploration versus Exploitation** in our Hill-Climber

- optimization algorithms which make largely local improvements are **exploiting** the local gradient, and algorithms which mostly wander about randomly are thought to **explore** the space

- as a rule of thumb: we prefer to use a highly exploitative algorithm (it's fastest), but the "uglier" (less structured, more random) the space, the more we will have no choice but to use a more explorative algorithm

Single-State Global Optimization Algorithms

- a **global optimization algorithm** is one which, if we run it long enough, will eventually find the global optimum

- almost always, the way this is done is by guaranteeing that, at the limit, every location in the search space will be visited

- the single-state algorithms we've seen so far cannot guarantee this - our version of Tweak does not give us the tools for getting from a sufficiently broad local optimum

- thus, the algorithms so far have been **local optimization algorithms**

- there are many ways to construct a global optimization algorithm instead, with the simplest one being **Random Search**

- random Search is the extreme in exploration (and global optimization); in contrast, Hill Climbing, with Tweak set to just make very small changes and never make large ones, may be viewed as the extreme in exploitation (and local optimization)

Algorithm 6 Random Search

$Best \leftarrow$ some initial random candidate solution
repeat
    $S \leftarrow$ a random candidate solution
    if Quality$(S) >$ Quality$(Best)$ **then**
        $Best \leftarrow S$
until $Best$ is the ideal solution or we have run out of time
return $Best$

- consider the following popular technique, called **Hill-Climbing with Random Restarts**, half-way between the two:
    - we do Hill-Climbing for a certain random amount of time
    - then when time is up, we start over with a new random location and do Hill-Climbing again for a different random amount of time
    - and so on

Algorithm 7 Hill-Climbing with Random Restarts

$T \leftarrow$ distribution of possible time intervals

$S \leftarrow$ initial candidate solution
$Best \leftarrow S$
repeat
   $time \leftarrow$ random time in the near future, chosen from T
   repeat
      $R \leftarrow \text{Tweak}(\text{Copy}(S))$
      if $\text{Quality}(R) > \text{Quality}(S)$ **then**
         $S \leftarrow R$
   until S is the ideal solution, or **time** is up, or we have run out of total time
   if $\text{Quality}(S) > \text{Quality}(Best)$ **then**
      $Best \leftarrow S$
   $S \leftarrow$ some random candidate solution
until $Best$ is the ideal solution or we have run out of total time
return $Best$
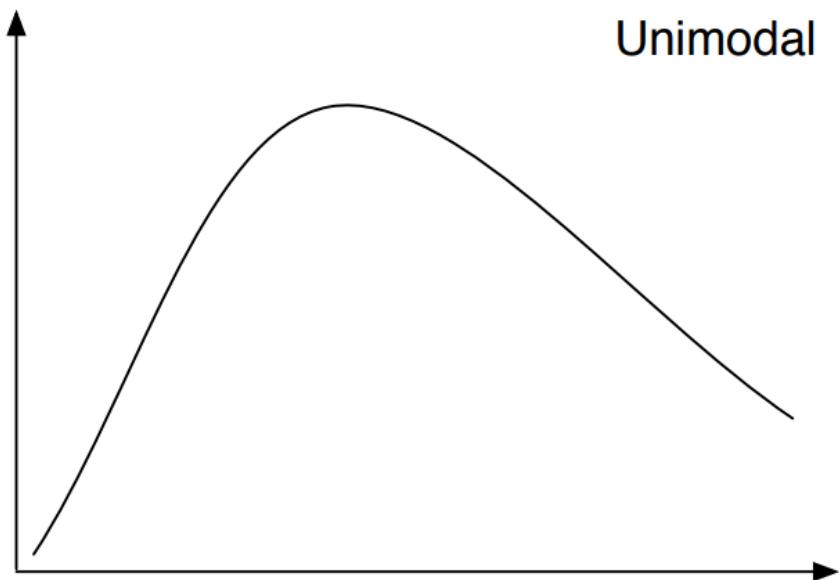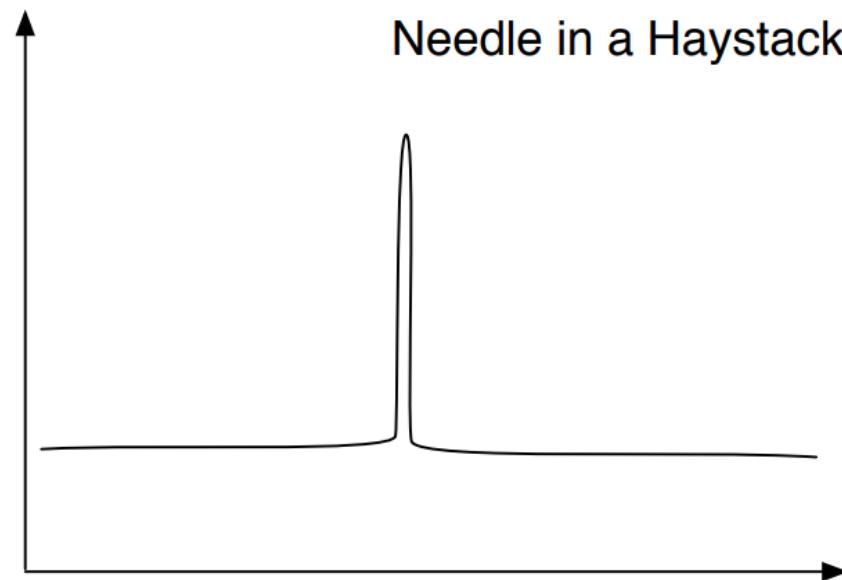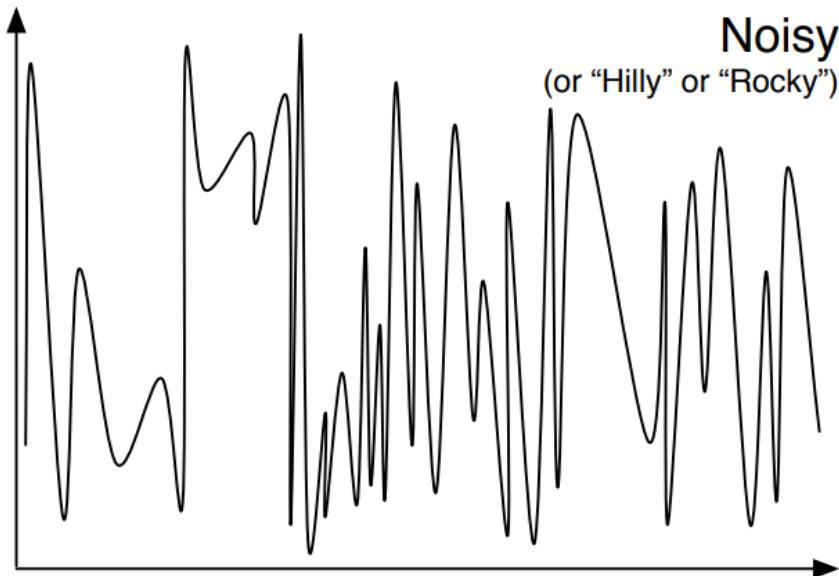
- if the randomly-chosen time intervals are generally extremely long, this algorithm is basically one big Hill-Climber

- likewise, if the intervals are very short, we're basically doing random search (by resetting to random new locations each time)

- moderate interval lengths run the gamut between the two - which is good, right?

- consider Figure on the next slide:

  – the first figure, labeled Unimodal, is a situation where Hill-Climbing is close to optimal, and where Random Search is a very bad pick

  – for the figure labelled Noisy, Hill-Climbing is quite bad; and in fact Random Search is expected to be about as good as you can do (not knowing anything about the functions beforehand)
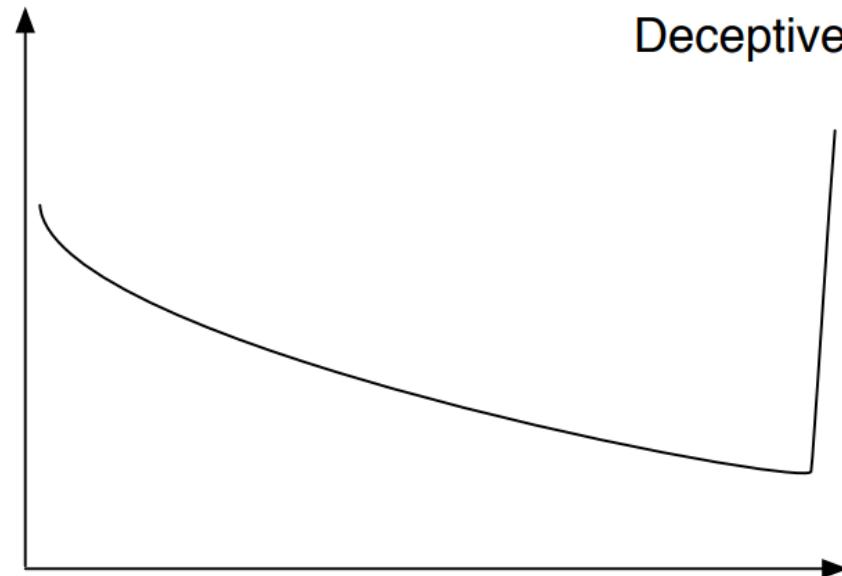
Unimodal

Needle in a Haystack

Noisy
(or "Hilly" or "Rocky")

Deceptive

– in the Noisy situation, there's no relationship like this: even similar solutions are very dissimilar in quality - this is often known as the **smoothness** criterion for local search to be effective

– consider the figure labeled Needle in a Haystack, for which Random Search is the only real way to go, and Hill-Climbing is quite poor - what's the difference between this and Unimodal

– for local search to be effective there must be an **informative gradient** which generally leads towards the best solutions - in fact, we can make highly uninformative gradients for which Hill-Climbing is spectacularly bad

– in the figure labeled Deceptive, Hill-Climbing not only will not easily find the optimum, but it is actively let away from the optimum

Simulated Annealing

- Simulated Annealing was developed by various researchers in the mid 1980s

- the algorithm varies from Hill-Climbing in its decision of when to replace S, the original candidate solution, with R, its newly tweaked child

  - if R is better than S, we'll always replace S with R as usual
  - if R is worse than S, we may still replace S with R with a certain probability $P(t, R, S)$:
  $$P(t, R, S) = e^{\frac{\text{Quality}(R) - \text{Quality}(S)}{t}},$$

  where $t \geq 0$, i.e. if R isn't much worse than S, we'll still select R with a reasonable probability

- we have a new tunable parameter t - if t is close to 0, the fraction is a large number, and so the probability is close to 0; if t is high, the probability is close to 1

- the idea is to initially set t to a high number, which causes the algorithm to move to every newly-created solution regardless of how good it is - we're doing a **random walk** in the space

- then t decreases slowly, eventually to 0, at which point the algorithm is doing nothing more than plain Hill-Climbing

- the rate at which we decrease t is called the algorithm's **schedule**

Algorithm 8 Simulated Annealing

$t \leftarrow$ temperature, initially a high nubmer

$S \leftarrow$ initial candidate solution
$Best \leftarrow S$
repeat
   $R \leftarrow \text{Tweak}(\text{Copy}(S))$
   if $\text{Quality}(R) > \text{Quality}(S)$ or a random number chosen from $[0,1] < e^{\frac{\text{Quality}(R)-\text{Quality}(S)}{t}}$ **then**
     $S \leftarrow R$
   Decrease t
   if $\text{Quality}(S) > \text{Quality}(Best)$ **then**
     $Best \leftarrow S$
until $Best$ is the ideal solution or we have run out of time
return $Best$

Tabu Search

- Tabu Search (developed by Fred Glover) employs a different approach to doing exploration: it keeps around a history of recently considered candidate solutions (known as the tabu list) and refuses to return to those candidate solutions until they're sufficiently far in the past

- the simplest approach to Tabu Search is to maintain a **tabu list** L, of some maximum length l, of candidate solutions we've seen so far

- whenever we adopt a new candidate solution, it goes in the tabu list - if the tabu list is too large, we remove the oldest candidate solution and it's no longer taboo to reconsider

- in the version on the next slide, we generate n tweaked children, but only consider the ones which aren't presently taboo

Algorithm 9 Tabu Search

$l \leftarrow$ desired maximum tabu list length
$n \leftarrow$ number of tweaks desired to sample the gradient

$S \leftarrow$ initial candidate solution
$Best \leftarrow S$
$L \leftarrow \{\}$ a tabu list of maximum length l $\triangleright$ impemented as FIFO queue
Enqueue S into L
repeat
   if Length$(L) > l$ **then**
      Remove oldest element from L
   $R \leftarrow$ Tweak(Copy(S))
   for $n - 1$ times **do**
      $W \leftarrow$ Tweak(Copy(S))
      if $W \notin L$ and (Quality$(W) >$ Quality(R) or $R \in L$) **then**
         $R \leftarrow W$
      if $R \notin L$ **then**
         $S \leftarrow R$
         Enqueue R into L
   if Quality$(S) >$ Quality$(Best)$ **then**
      $Best \leftarrow S$
until $Best$ is the ideal solution or we have run out of time
return $Best$

- Tabu Search really only works in discrete spaces (although there are modifications for real-valued ones)

- the big problem with Tabu Search is that if your search space is very large, and particularly if it's of high dimensionality, it's easy to stay around in the same neighborhood, indeed on the same hill, even if you have a very large tabu list

- an alternative approach is to create a tabu list not of candidate solutions you've considered before, but of changes you've made recently to certain features (implementation details are in the book)

Iterated Local Search

- Iterated Local Search (ILS) is essentially a more clever version of Hill-Climbing with Random Restarts

- assuming we give it enough time between restarts, whenever we do a random restart the hill-climber winds up in some (possibly new) local optimum

- thus we can think of Hill-Climbing with Random Restarts as doing a sort of random search through the space of local optima

- we find a random local optimum, then another, then another, and so on, and eventually return the best optimum we ever discovered (ideally, it's a global optimum!)

- ILS tries to search through this space of local optima in a more intelligent fashion: it tries to stochastically hill-climb in the space of local optima

- ILS finds a local optimum, then looks for a "nearby" local optimum and possibly adopts that one instead, then finds a new "nearby" local optimum, and so on

- the heuristic here is that you can often find better local optima near to the one you're presently in, and walking from local optimum to local optimum in this way often outperforms just trying new locations entirely at random

- ILS doesn't pick new restart locations entirely at random - rather, it maintains a "home base" local optimum of sorts, and selects new restart locations that are somewhat, though not excessively, in the vicinity of the "home base" local optimum

- we want to restart far enough away from our current home base to wind up in a new local optimum, but not so far as to be picking new restart locations essentially at random (we want to be doing a walk rather than a random search)

- when ILS discovers a new local optimum, it decides whether to retain the current "home base" local optimum, or to adopt the new local optimum as the "home base"

- if we always pick the new local optimum, we're doing a random walk (a sort of meta-exploration)

- if we only pick the new local optimum if it's better than our current one, we're doing hill-climbing (a sort of meta-exploitation)

- the algorithm is relatively straightforward: do hill-climbing for a while; then (when time is up) determine whether to adopt the newly discovered local optimum or to retain the current "home base" one (the NewHome-Base function); then from our new home base, make a very big Tweak (the Perturb function), which is ideally just large enough to likely jump to a new hill

Algorithm 10 Iterated Local Search (ILS) with Random Restarts

$T \leftarrow$ distribution of time intervals

$S \leftarrow$ initial random candidate solution
$H \leftarrow S$
$Best \leftarrow S$
repeat
   $time \leftarrow$ random time in the near future, chosen from T
   repeat
      $R \leftarrow \text{Tweak}(\text{Copy}(S))$
      if $\text{Quality}(R) > \text{Quality}(S)$ **then**
         $S \leftarrow R$
   until S is the ideal solution, or $time$ is up, or we have run out of total time
   if $\text{Quality}(S) > \text{Quality}(Best)$ **then**
      $Best \leftarrow S$
   $H \leftarrow \text{NewHomeBase}(H, S)$
   $S \leftarrow \text{Perturb}(H)$
until $Best$ is the ideal solution or we have run out of total time
return $Best$

- much of the thinking behind the choices of Perturb and NewHomeBase functions is a black art, determined largely by the nature of the particular problem being tackled

- the goal of the Perturb function is to make a very large Tweak, big enough to likely escape the current local optimum, but not so large as to be essentially a randomization (we'd like to fall onto a nearby hill)

- the goal of the NewHomeBase function is to intelligently pick new starting locations - at one extreme, the algorithm could always adopt the new local optimum, that is,

$$\text{NewHomeBase}(H, S) = S,$$

which results in essentially a random walk from local optimum to local optimum

- at the other extreme, the algorithm could only use the new local optimum if it's of equal or higher quality than the old one, that is,

$$\text{NewHomeBase}(H, S) = \begin{cases} S, & \text{if Quality}(S) \geq \text{Quality}(H), \\ H, & \text{otherwise}, \end{cases}$$

which results in a kind of hill-climbing among the local optima

- most ILS heuristics try to strike a middle-ground between the two - for example, ILS might hill-climb unless it hasn't seen a new and better solution in a while, at which point it starts doing random walks for a bit

- there are other options of course: we could apply a Simulated Annealing approach to NewHomeBase, or a Tabu Search procedure of sorts

- the algorithms described thus far are not set in stone, as there are lots of ways to mix and match them, or develop other approaches entirely

Population Methods

- population-based methods differ from the previous methods in that they keep around a sample of candidate solutions rather than a single candidate solution

- each of the solutions is involved in tweaking and quality assessment, but what prevents this from being just a parallel hill-climber is that candidate solutions affect how other candidates will hill-climb in the quality function

- this could happen either by good solutions causing poor solutions to be rejected and new ones created, or by causing them to be Tweaked in the direction of the better solutions

- most population-based methods steal concepts from biology - one particularly popular set of techniques, collectively known as **Evolutionary Computation** (EC), borrows liberally from population biology, genetics, and evolution

- an algorithm chosen from this collection is known as an **Evolutionary Algorithm** (EA), these can be divided into:

  - **generational algorithms** - update the entire sample once per iteration

  - **steady-state algorithms** - update the sample a few candidate solutions at a time

- common EAs include the Genetic Algorithm (GA) and Evolution Strategies (ES); and there are both generational and steady-state versions of each

- because they are inspired by biology, EC methods tend to use (and abuse) terms from genetics and evolution

- Common terms used in EC:

individual	a candidate solution
child and parent	a child is the Tweaked copy of a candidate solution (its parent)
population	set of candidate solutions
fitness	quality
fitness landscape	quality function
fitness assessment or evaluation	computing the fitness of an individual
selection	picking individuals based on their fitness
mutation	plain Tweaking - this is often thought as "asexual" breeding
recombination or crossover	a special Tweak which takes two parents, swaps sections of them, and (usually) produces two children - this is often thought as "sexual" breeding
breeding	producing one or more children from a population of parents through an iterated process of selection and Tweaking (typically mutation or recombination)
genotype or genome	an individual's data structure, as used during breeding
chromosome	a genotype in the form of a fixed-length vector
gene	a particular slot position in a chromosome
allele	a particular setting of a gene
phenotype	how the individual operates during fitness assessment
generation	one cycle of fitness assessment, breeding, and population reassembly; or the population produced each such cycle

- EC techniques are generally resampling techniques: new samples (populations) are generated or revised based on the results from older ones

- in contrast, Particle Swarm Optimization (which we have seen in the previous course), is an example of a directed mutation method, where candidate solutions in the population are modified, but no resampling occurs per se

- The basic generational evolutionary computation algorithm first constructs an initial population, then iterates through three procedures:

  - first, it **assesses the fitness** of all the individuals in the population

  - second, it uses this fitness information to breed a new population of children

  - third, it **joins** the parents and children in some fashion to form a new next-generation population, and the cycle continues

---

Algorithm 11 An Abstract Generational Evolutionary Algorithm (EA)

$P \leftarrow$ Build Initial Population
$Best \leftarrow \square$ ▷ $\square$ means "nobody yet"
repeat
    AssessFitness(P)
    for each individual $P_i \in P$ **do**
       if $Best = \square$ or Fitness(P_i) > Fitness($Best$) **then**   ▷ Fitness is just Quality
          $Best \leftarrow P_i$
       $P \leftarrow$ Join(P,Breed(P))
until $Best$ is the ideal solution or we have run out of time
return $Best$

- unlike the Single-State methods, we have a separate AssessFitness function; we need the fitness values of our individuals before we can Breed them

- EAs differ largely in how they perform the Breed and Join operations

- Breed operation usually has two parts: **Selecting** parents from the old population, then Tweaking them (usually **Mutating** or **Recombining** them in some way) to make children

- Join operation either completely replaces the parents with the children, or includes fit parents along with their children to form the next generation

Evolution Strategies

- the family of algorithms known as Evolution Strategies (ES) were developed by Ingo Rechenberg and Hans-Paul Schwefel in the mid 1960s

- ES employ a simple procedure for selecting individuals called **Truncation Selection**, and (usually) only uses mutation as the Tweak operator

- among the simplest ES algorithms is the (μ, λ) algorithm - we begin with a population of (typically) λ number of individuals, generated randomly, then, proceed as follows:

  - first, we assess the fitness of all the individuals

  - then we delete from the population all but the μ fittest ones (this is all there's to Truncation Selection)

  - each of the μ fittest individuals gets to produce λ/μ children through an ordinary Mutation, creating λ new childred in total

  - our Join operation is simple: the children just replace the parents, who are discarded (and go back to step one)
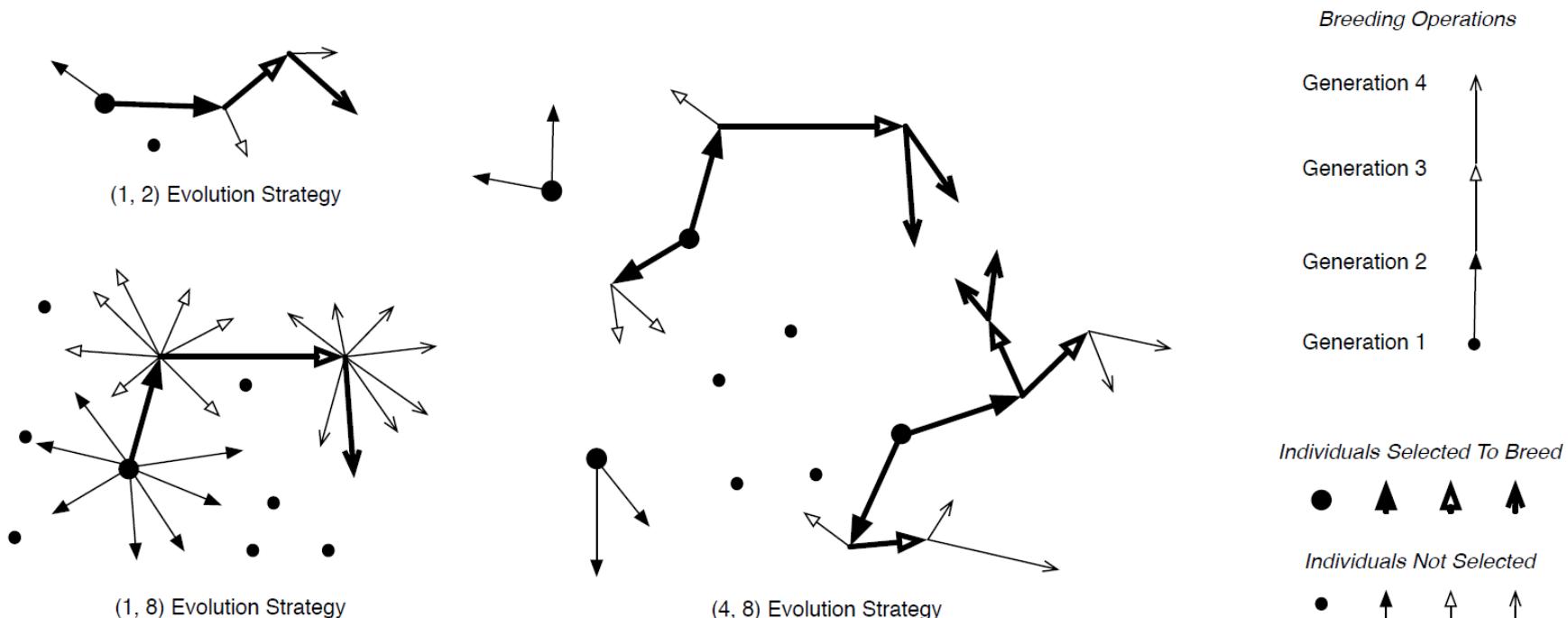
Algorithm 12 The (μ, λ) Evolution Strategy

$\mu \leftarrow$ number of parents selected
$\lambda \leftarrow$ number of children generated by the parents

$P \leftarrow \{\}$
for λ times **do**
   $P \leftarrow P \cup \{\text{new random individual}\}$
$Best \leftarrow \square$
repeat
   for each individual $P_i \in P$ **do**
      AssessFitness(P_i)
      if $Best = \square$ or Fitness(P_i) > Fitness($Best$) **then**
         $Best \leftarrow P_i$
   $Q \leftarrow$ the μ individuals in P whose Fitness() are greatest   ▷ Truncation Selection
   $P \leftarrow \{\}$ ▷ Join is done by just replacing P with the children
   for each individual in $Q_j \in Q$ **do**
      for λ/μ times **do**
         $P \leftarrow P \cup \{\text{Mutate(Copy}(Q_j))\}$
until $Best$ is the ideal solution or we have run out of time
return $Best$

- notice that λ should be a multiple of μ

- ES practitioners usually refer to their algorithm by the choice of μ and λ (i.e., if $\mu = 5$ and $\lambda = 20$, then we have a "(5, 20) Evolution Strategy")

- the (μ, λ) algorithm has three knobs with which we may adjust exploration versus exploitation:

  - the size of λ - controls the sample size for each population, and is basically the same thing as the n variable in Steepest-Ascent Hill Climbing With Replacement

  - a the extreme, as λ approaches ∞, the algorithm approaches exploration (random search)

  - the size of μ - controls how selective the algorithm is; low values of μ with respect to λ push the algorithm more towards exploitative search as only the best individuals survive

  - the degree to which Mutation is performed - if Mutate has a lot of noise, then new children fall far from the tree and are fairly random regardless of the selectivity of μ

Breeding Operations

(1, 2) Evolution Strategy

(1, 8) Evolution Strategy

(4, 8) Evolution Strategy

Generation 4

Generation 3

Generation 2

Generation 1

Individuals Selected To Breed

Individuals Not Selected

- the second Evolution Strategy algorithm is called $(\mu + \lambda)$ - it differs from (μ, λ) in only one respect: the Join operation

- in (μ, λ) the parents are simply replaced with the children in the next generation

- in $(\mu + \lambda)$, the next generation consists of the μ parents plus the λ new children

Algorithm 13 The $(\mu + \lambda)$ Evolution Strategy

$\mu \leftarrow$ number of parents selected
$\lambda \leftarrow$ number of children generated by the parents

$P \leftarrow \{\}$
for λ times **do** $\triangleright$ sometimes $\mu + \lambda$
    $P \leftarrow P \cup \{$new random individual$\}$
$Best \leftarrow \square$
repeat
    for each individual $P_i \in P$ **do**
        AssessFitness(P_i)
        if $Best = \square$ or Fitness(P_i) > Fitness($Best$) **then**
            $Best \leftarrow P_i$
    $Q \leftarrow$ the μ individuals in P whose Fitness() are greatest $\triangleright$ Truncation Selection
    $P \leftarrow Q$ $\triangleright$ The Join operation is the only difference with (μ, λ)
    for each individual in $Q_j \in Q$ **do**
        for λ/μ times **do**
            $P \leftarrow P \cup \{$Mutate(Copy(Q_j))$\}$
until $Best$ is the ideal solution or we have run out of time
return $Best$

- the parents compete with the kids next time around - the next and all successive generations are $\mu + \lambda$ in size

- generally speaking, $\mu + \lambda$ may be more exploitative than (μ, λ) because high-fitness parents persist to compete with the children
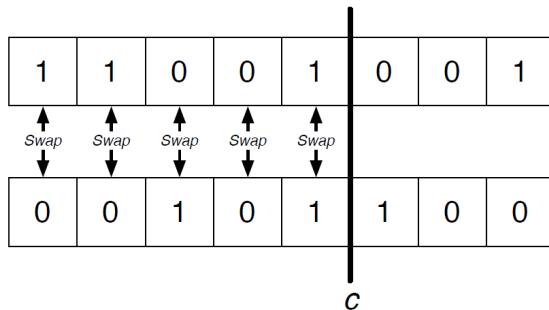
Genetic Algorithms

- Genetic Algorithms (GAs), were invented by John Holland in the 1970s

- the algoritm is similar to the (μ, λ) ES in many respects: it iterates through fitness assessment, selection and breeding, and population reassembly

- the primary difference is in how selection and breeding take place: whereas ES select all the parents and then create all the children, GA little-by-little selects a few parents and generates a few children until enough children have been created

- to breed, we begin with an empty population of children; we then select two parents from the original population, copy them, cross them over with one another, and mutate the results

- this forms two children, which we then add to the child population; we repeat this process until the child population is entirely filled
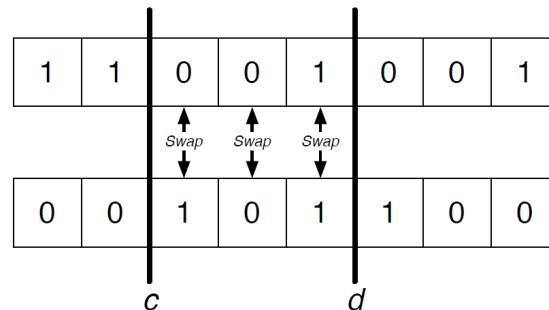
Algorithm 14 The Genetic Algorithm (GA)

$popsize \leftarrow$ desired population size $\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ ▷ This is basically λ. Make it even.

$P \leftarrow \{\}$
for λ times **do**
$\quad P \leftarrow P \cup \{\text{new random individual}\}$
$Best \leftarrow \square$
repeat
$\quad$ **for** each individual $P_i \in P$ **do**
$\quad\quad$ AssessFitness(P_i)
$\quad\quad$ **if** $Best = \square$ or Fitness(P_i) > Fitness($Best$) **then**
$\quad\quad\quad Best \leftarrow P_i$
$\quad Q \leftarrow \{\}$ $\qquad\qquad\qquad\qquad\qquad\qquad$ ▷ Here's where we begin to deviate from (μ, λ)
$\quad$ **for** $popsize/2$ times **do**
$\quad\quad$ Parent $P_a \leftarrow$ SelectWithReplacement(P)
$\quad\quad$ Parent $P_b \leftarrow$ SelectWithReplacement(P)
$\quad\quad$ Childred $C_a, C_b \leftarrow$ Crossover(Copy(P_a),Copy(P_b))
$\quad\quad Q \leftarrow Q \cup \{\text{Mutate}(C_a), \text{Mutate}(C_b)\}$
$\quad P \leftarrow Q$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ ▷ End of deviation
until $Best$ is the ideal solution or we have run out of time
**return** $Best$
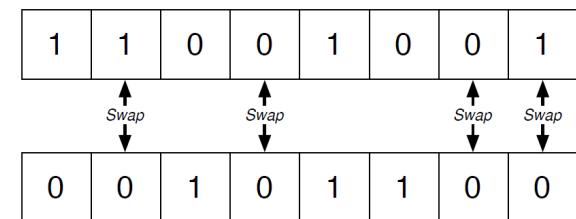
- note how similar the Genetic Algorithm is to (μ, λ), except during the breeding phase

- to perform breeding, we need two new functions we've not seen before: SelectWithReplacement and Crossover; plus of course Mutate

1	1	0	0	1	0	0	1

Swap Swap Swap Swap Swap

0	0	1	0	1	1	0	0

c

One-Point Crossover.

1	1	0	0	1	0	0	1

Swap Swap Swap

0	0	1	0	1	1	0	0

c d

Two-Point Crossover.

1	1	0	0	1	0	0	1

Swap Swap Swap Swap

0	0	1	0	1	1	0	0
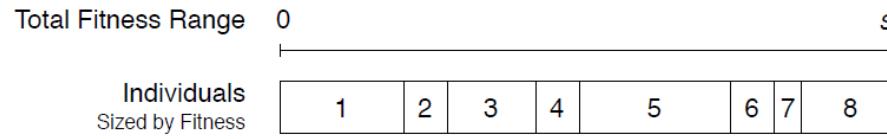
Uniform Crossover.

- we center our attention on the genome being a boolean vector (i.e., variable in binary optimization problems, like the Knapsack we have seen throughout the course)

- one simple way is **bit-flip mutation**: march down the vector, and flip a coin of a certain probability (often $1/l$, where l is the length of the vector)

- **crossover** is the GA's distinguishing feature; it involves mixing and matching parts of two parents to form children (how we do that mixing and-matching depends on the representation of the individuals)

- there are three classic ways of doing crossover in vectors: **One-Point**, **Two-Point**, and **Uniform Crossover**

313

- crossover is not a global mutation - if we cross over two vectors you can't produce any conceivable vector

- far we've been doing crossovers that are just swaps: but if the vectors are of floating-point values, our recombination could be something fuzzier, like averaging the two values rather than swapping them

- these operations can also be defined on other problem representations (graph structures, sets, trees, etc.)

- regarding **selection** - in ES, we just lopped off all but the m best individuals, a procedure known as Truncation Selection

- because the GA performs iterative selection, crossover, and mutation while breeding, we have more options

- unlike Truncation Selection, the GA's SelectWithReplacement procedure can (by chance) pick certain Individuals over and over again, and it also can (by chance) occasionally select some low-fitness Individuals

- the original SelectWithReplacement technique for GAs was called Fitness-Proportionate Selection, sometimes known as Roulette Selection. In this algorithm, we select individuals in proportion to their fitness: if an individual has a higher fitness, it's selected more often

- let $s = \sum f_i$ be the sum fitness of all the individuals; a random number from 0 to s falls within the range of some individual, which is then selected

- there is a relatively big problem with this method: it presumes that the actual fitness value of an individual really means something important, but often we choose a fitness function such that higher ones are "better" than smaller ones, and don't mean to imply anything more

- consider the following situation: a fitness function goes from 0 to 10, but, near the end of a run, all the individuals have values like 9.98, 9.99, etc.

Algorithm 15 Tournament Selection

$P \leftarrow$ population
$t \leftarrow$ tournament size, $t \geq 1$

$Best \leftarrow$ individual picked at random from P with replacement
for i from 2 to t **do**
   $Next \leftarrow$ individual picked at random from P with replacement
   if Fitness($Next$) $>$ Fitness($Best$) **then**
     $Best \leftarrow Next$
return $Best$

- we want to finesse the peak of the fitness function, and so we want to pick the 9.99-fitness individual, but to Fitness-Proportionate Selection, all these individuals will be selected with nearly identical probability

- to fix this we could **scale** the fitness function to be more sensitive to the values at the top end of the function

- to remedy the situation we need to adopt a **non-parametric** selection algorithm which throws away the notion that fitness values mean anything other than bigger is better, and just considers their rank ordering

- Truncation Selection does this, but the most popular technique by far is Tournament Selection, an astonishingly simple algorithm

- we return the fittest individual of some t individuals picked at random, with replacement, from the population

- Tournament Selection has become the primary selection technique used for the GAs and many related methods, for several reasons:

  - it's not sensitive to the particulars of the fitness function

  - it's simple, requires no preprocessing, and works well with parallel algorithms

  - it's tunable: by setting the tournament size t, we can change how selective the technique is; at the extremes, if $t = 1$, this is just random search

  - if t is very large (much larger than the population size itself), then the probability that the fittest individual in the population will appear in the tournament approaches 1.0, and so Tournament Selection just picks the fittest individual each time

Elitism

- in Elitism, we augment the Genetic Algorithm to directly inject into the next population the fittest individual or individuals from the previous population, called the **elites**

- by keeping the best individual (or individuals) around in future populations, this algorithm begins to resemble $(\mu + \lambda)$, and has similar exploitation properties

- this exploitation can cause premature convergence if not kept in check: perhaps by increasing the mutation and crossover noise, or weakening the selection pressure, or reducing how many elites are being stored

- a minor catch - if we want to maintain a population size of popsize, and you're doing crossover, we'll need to have popsize, minus the number of elites, be divisible by two

Hybrid Optimization Algorithms

- there are many ways in which we can create hybrids of various meta-heuristics algorithms, but perhaps the most popular approach is a hybrid of evolutionary computation and a local improver such as hill-climbing

- the EA could go in the inner loop and the hill-climber outside: for example, we could extend Iterated Local Search (ILS, Section 2.6) to use a population method in its inner loop, rather than a hill-climber, but retain the "Perturb" hill-climber in the outer loop

- but by far the most common method is the other way around: augment an EA with some hill-climbing during the fitness assessment phase to revise each individual as it is being assessed

Combinatorial Optimization

- a combinatorial optimization problem is one in which the solution consists of a combination of unique components selected from a typically finite, and often small, set, with the objective to find the optimal combination of components

- one example is the knapsack problem we have seen several times already

- another example is the classic **traveling salesman problem** (or TSP), which has a set of cities with some number of routes (plane flights, say) between various pairs cities

- in TSP, each route has a cost, the salesman must construct a tour starting at city A, visiting all the cities at least once, and finally, returning to A woith lowest possible cost

- Knapsack does have one thing the TSP doesn't have: it has additional weights (of the item) and a maximum "Weight" which must not be exceeded

- the TSP has a different notion of infeasible solutions than simply ones which exceed a certain bound

- combinatorial optimization problems can be solved by most general-purpose metaheuristics such as those we've seen so far, and in fact certain techniques (Iterated Local Search, Tabu Search, etc.) are commonly promoted as combinatorics problem methods

- but some care must be taken because most metaheuristics are really designed to search much more general, wide-open spaces than the constrained ones found in most combinatorial optimization problems

- as an example, consider the use of a boolean vector in combination with a metaheuristic such as simulated annealing or the genetic algorithm

- each slot in the vector represents a component (i.e., knapsack item), and if the slot is true, then the component is used in the candidate solution

- the problem with this approach is that it's easy to create solutions which are infeasible

- in the knapsack problem we have declared that solutions which are larger than the knapsack are simply illegal

- in Knapsack, it's not a disaster to have candidate solutions like that, as long as the final solution is feasible—we could just declare the quality of such infeasible solutions to be their distance from the optimum (in this case perhaps how overfull the knapsack is); we might punish them further for being infeasible

- but in a problem like the Traveling Salesman Problem, our boolean vector might consist of one slot per edge in the TSP graph

- it's easy to create infeasible solutions for the TSP which are simply non-sense: how do we assess the "quality" of a candidate solution whose TSP solution isn't even a tour

- the issue here is that these kind of problems, as configured, have **hard constraints**: there are large regions in the search space which are simply invalid

- ultimately we want a solution which is feasible; and during the search process it'd be nice to have feasible candidate solutions so we can actually think of a way to assign them quality assessment

- there are two parts to this: initialization (construction) of a candidate solution from scratch, and Tweaking a candidate solution into a new one

- **Construction:** iterative construction of components within hard constraints is sometimes straightforward and sometimes not; it's often done like this

  1. Choose a component. For example, in the TSP, pick an edge between two cities A and B. In Knapsack, it's an initial item. Let our current (partial) solution start with just that component.

  2. Identify the subset of components that can be concatenated to components in our partial solution. In the TSP, this might be the set of all edges going out of A or B. In Knapsack, this is all items that can still be added into the knapsack without going over.

  3. Tend to discard the less desirable components. In the TSP, we might emphasize edges that are going to cities we've not visited yet if possible.

  4. Add to the partial solution a component chosen from among those components not yet discarded.

  5. Quit when there are no components left to add. Else go to step 2.

- **Tweaking:** the Tweak operator can be even harder to do right, because in the solution space feasible solutions may be surrounded on all sides by infeasible ones; four common approaches are:

  – Invent a closed Tweak operator which automatically creates feasible children. This can be a challenge to do, particularly if you're including crossover. And if you create a closed operator, can it generate all possible feasible children? Is there a bias? Do you know what it is?

  – Repeatedly try various Tweaks until you create a child which is feasible. This is relatively easy to do, but it may be computationally expensive.

  – Allow infeasible solutions but construct a quality assessment function for them based on their distance to the nearest feasible solution or to the optimum. This is easier to do for some problems than others. For example, in the Knapsack problem it's easy: the quality of an overfull solution could be simply based on how overfull it is (just like underfull solutions).

– Assign infeasible solutions a poor quality. This essentially eliminates them from the population; but of course it makes your effective population size that much smaller. It has another problem too: moving just over the edge between the feasible and infeasible regions in the space results in a huge decrease in quality. In Knapsack, for example, the best solutions are very close to infeasible ones because they're close to filled. So one little mutation near the best solutions and whammo, you're infeasible and have big quality punishment. This makes optimizing near the best solutions a bit like walking on a tightrope.

- none of these is particularly inviting

- while it's often easy to create a valid construction operator, making a good Tweak operator that's closed can be pretty hard

Component-Oriented Methods

- these methods work by taking advantage of the fact that the solutions in these spaces consist of combinations of components drawn from a typically fixed set

- it's the presence of this fixed set that we can take advantage of in a greedy, local fashion by maintaining historical "quality" values, so to speak, of individual components rather than (or in addition to) complete solutions

- there are two reasons you might want to do this:

  - while constructing, to tend to select from components which have proven to be better choices

  - while Tweaking, to modify those components which appear to be getting us in a local optimum

Greedy Randomized Adaptive Search Procedures

- GRASP, by Thomas Feo and Mauricio Resende, is a single-state meta-heuristic, which is built on the notions of constructing and Tweaking feasible solutions

- the overall algorithm is really simple: we create a feasible solution by constructing from among highest value (lowest cost) components and then do some hill-climbing on the solution (pseudocode on the next slide)

- instead of picking the $p\%$ best available components, some versions of GRASP pick components from among the components whose value is no less than (or cost is no higher than) some amount

- GRASP is more or less using a truncation selection among components to do its initial construction of candidate solutions

Algorithm 16 Greedy Randomized Adaptive Search Procedures (GRASP)

$C \leftarrow \{C_1, \ldots, C_n\}$ components
$p \leftarrow$ percentage of components to include each iteration
$m \leftarrow$ length of time to do hill-climbing

$Best \leftarrow \square$
repeat
  $S \leftarrow \{\}$
  repeat
    $C' \leftarrow$ components in $C - S$ which could be added to S without being infeasible
    if C' is empty **then**
      $S \leftarrow \{\}$
    else
      $C'' \leftarrow$ the $p\%$ highest value (or lowest cost) components in C'
      $S \leftarrow S \cup \{$component chosen uniformly at random from $C''\}$
  until S is a complete solution
  for m times **do**
    $R \leftarrow \text{Tweak}(\text{Copy}(S))$ ▷ Tweak must be closed, that is, it must create feasible solutions
    if $\text{Quality}(R) > \text{Quality}(S)$ **then**
      $S \leftarrow R$
  if $Best = \square$ or $\text{Quality}(S) > \text{Quality}(Best)$ **then**
    $Best \leftarrow S$
until $Best$ is the ideal solution or we have run out of time
return $Best$

- GRASP shows how to construct candidate solutions by iteratively picking components; but it's still got the same problem as EC methods when it comes to the Tweak step: we have to find a way of guaranteeing closure

Ant Colony Optimization

- Marco Dorigo's Ant Colony Optimization (or ACO) is an approach to combinatorial optimization which gets out of the issue of Tweaking by making it optional; rather, it simply assembles candidate solutions by selecting components which compete with one another for attention

- ACO is population-oriented, but there are two different kinds of "populations" in ACO

- first, there is the set of components that make up a candidate solutions to the problem: in the Knapsack problem this set would consist of all the items; in the TSP, it would consist of all the edges

- the set of components never changes: but we will adjust the "fitness" (called the **pheromone**) of the various components in the population as time goes on

Algorithm 17 An Abstract Ant Colony Optimization Algorithm (ACO)

$C \leftarrow \{C_1, \ldots, C_n\}$ components
$popsize \leftarrow$ number of trails to build at once   ▷ "ant trails" are "candidate solutions"

$p \leftarrow \langle \rangle$ pheromones of the components, initially zero
$Best \leftarrow \square$
repeat
    $P \leftarrow popsize$ trails built by iteratively selecting components based on pheromones and costs or values
    for $P_i \in P$ **do**
        $P_i \leftarrow$ Optionally Hill-Climp P_i
        if $Best = \square$ or Fitness(P_i) > Fitness($Best$) **then**
            $Best \leftarrow P_i$
    Update p for components based on the fitness results for each $P_i \in P$ in which they participated
until $Best$ is the ideal solution or we have run out of time
return $Best$

- each generation we build one or more candidate solutions, called **ant trails**, by selecting components one by one based, in part, on their pheromones - this constitutes the second "population" in ACO: the collection of trails

- then we assess the fitness of each trail; for each trail, each of the components in that trail is then updated based on that fitness: a bit of the trail's fitness is rolled into each component's pheromone

- both ACO and GRASP iteratively build candidate solutions, then hill-climb them, but there are obvious differences:

  - ACO builds some *popsize* candidate solutions all at once

  - ACO's hill-climbing is optional, and indeed it's often not done at all

  - if we're finding it difficult to construct a closed Tweak operator for our particular representation, we can entirely skip the hill-climbing step if need be

  - most importantly, components are selected not just based on component value or cost, but also on pheromones; pheromone is essentially the "historical quality" of a component: often approximately the sum total (or mean, etc.) fitness of all the trails that the component has been a part of

- pheromones tell us how good a component would be to select regardless of its (possibly low) value or (high) cost

- after assessing the fitness of trails, we update the pheromones in some way to reflect new fitness values we've discovered so those components are more or less likely to be selected in the future

- the first version of ACO was the Ant System (AS); it's not used as often nowadays but is a good starting point to illustrate these notions

- in the Ant System, we select components based on a fitness-proportionate selection procedure of sorts, employing both costs or values and pheromones (we'll get to that); we then always add fitnesses into the component pheromones

- since this could cause the pheromones to go sky-high, we also always reduce (or **evaporate**) all pheromones a bit each time

- the Ant System has five basic steps:

  1. Construct some trails (candidate solutions) by selecting components.

  2. (Optionally) Hill-Climb the trails to improve them.

  3. Assess the fitness of the final trails.

  4. "Evaporate" all the pheromones a bit.

  5. Update the pheromones involved in trails based on the fitness of those solutions.

- in the original AS algorithm, there's no hill-climbing, but later versions of ACO include it

Algorithm 18 The Ant System (AS)

$C \leftarrow \{C_1, \ldots, C_n\}$ components
$e \leftarrow$ evaporation constant,$0 < e \leq 1$
popsize $\leftarrow$ number of trails to build at once
$\gamma \leftarrow$ initial value for pheromones
$t \leftarrow$ iterations to Hill-Climb

$p \leftarrow \langle \rangle$ pheromones of the components, all set to γ
$Best \leftarrow \square$
repeat
  $P \leftarrow \{\}$ ▷ Our trails (candidate solutions)
  for *popsize* times **do**
    $S \leftarrow \{\}$
    repeat
      $C' \leftarrow$ components in $C - S$ which could be added to S without being infeasible
      if C' is empty **then**
        $S \leftarrow \{\}$
      else
        $S \leftarrow S \cup \{$component selected from C' based on pheromones and values or costs$\}$
    until S is a complete trail
    $S \leftarrow$ Hill-Climb(S) for t iterations   ▷ Optinal. By default, not done.
    AssessFitness(S)
    if $Best = \square$ or Fitness$(S) >$ Fitness$(Best)$ **then**
      $Best \leftarrow S$
    $P \leftarrow P \cup \{S\}$
  for each $p_i \in p$ **do**   ▷ Decrease all pheromones a bit ("evaporation")
    $p_i \leftarrow (1 - e)p_i$
  for each $P_j \in P$ ▷ Update pheromones in components used in trails
    for each component C_i **do**
      if C_i was used in P_j **then**
        $p_i \leftarrow p_i +$ Fitness(P_j)
until $Best$ is the ideal solution or we have run out of time
return $Best$

- we construct trails by repeatedly selecting from those components which, if added to the trail, wouldn't make it infeasible

- Knapsack is easy: keep on selecting blocks until it's impossible to select one without going over

- but the TSP is more complicated; for example, in the TSP we could just keep selecting edges until we have a complete tour. But we might wind up with edges we didn't need, or a bafflingly complex tour

- other approach is to start with a city, then select from among those edges going out of the city to some city we've not seen yet (unless we have no choice), then select from among edges going out of that city, and so on

- AS selects using what we can call a component's desirability: combining values and pheromones:

$$\text{Desirability}(C_i) = p_i^{\delta} \times (\text{Value}(C_i))^{\epsilon},$$

where δ and ϵ are tuning parameters

- note that as the pheromone goes up the quality goes up; if a component has a higher value (or lower cost), then the quality goes up

- AS simply does a "desirability-proportionate" selection among the components we're considering, similar to fitness-proportionate selection

- we could perform some other selection procedure among our components, like tournament selection or GRASP-style truncation to $p\%$ based on desirability

- the Ant System evaporates pheromones because otherwise the pheromones keep on piling up, but there's perhaps a better way to do it: adjust the pheromones up or down based on how well they've performed on average.

- instead of evaporating and updating as was shown in the Ant System, we could just take each pheromone p_i and adjust it

Algorithm 19 Pheromone Updating with a Learning Rate

$C \leftarrow \{C_1, \ldots, C_n\}$ components
$p \leftarrow \langle p_1, \ldots, p_n \rangle$ pheromones for the components
$P \leftarrow \{P_1, \ldots, P_m\}$ population of trails
$\alpha \leftarrow$ learning rate

$r \leftarrow \langle r_1, \ldots, r_n \rangle$ total desirability of each component, initially 0
$c \leftarrow \langle c_1, \ldots, c_n \rangle$ component usage counts, initially 0
for each $P_j \in P$ **do**   ▷ Compute the average fitness of trails which employed each component
   for each component C_i **do**
      if C_i was used in P_j **then**
         $r_i \leftarrow r_i +$ Desirability(P_j)
         $c_i \leftarrow c_i + 1$
for each $p_i \in p$ **do**
   if $c_i > 0$ **then**
      $p_i \leftarrow (1 - \alpha)p_i + \alpha \frac{r_i}{c_i}$ ▷ $\frac{r_i}{c_i}$ is the average fitness computed earlier
return p

- $0 \leq \alpha \leq 1$ is the learning rate; for each component, we're computing the average fitness of every trail which used that component
- then we're throwing out a small amount of what we know so far ($1 - \alpha$'s worth) and rolling in a little bit of what we've just learned this iteration about how good a component is (α's worth)
- if α is large, we quickly adopt new information at the expense of our historical knowledge - it's probably best if α is small

- there have been a number of improvements on AS since it was first proposed, one particularly well-known one is the **Ant Colony System** (**ACS**)

- ACS works like the Ant System but with the following changes:

  - the use of an elitist approach to updating pheromones: only increase pheromones for components used in the best trail discovered so far; in a sense this starts to approach $(1 + \lambda)$

  - the use of a learning rate in pheromone updates

  - a slightly different approach for evaporating pheromones

  - a strong tendency to select components that were used in the best trail discovered so far

- ACS only improves the pheromones of components that were used in the best-sofar trail (the trail we store in $Best$), using the learning rate method shown on the previous slide

- That is, if a component is part of the best-so-far trail, we increase its pheromones as $p_i \leftarrow (1 - \alpha)p_i + \alpha\text{Fitness}(Best)$

- this is very strongly exploitative, so all pheromones are also decreased whenever they're used in a solution, notionally to make them less desirable for making future solutions in order to push the system to explore a bit more in solution space

- more specifically, whenever a component C_i is used in a solution, we adjust its pheromone $p_i \leftarrow (1 - \beta)p_i + \beta\gamma$, where β is a sort of evaporation or "unlearning rate", and γ is the value we initialized the pheromones to originally (left alone, this would eventually reset the pheromones to all be γ)

- component selection is also pretty exploitative - we flip a coin of probability q: if it comes up heads, we select the component which has the highest Desirability; otherwise we select in the same way as AS selected (though ACS simplifies the selection mechanism by getting rid of δ)

Algorithm 20 The Ant Colony System (ACS)

$C \leftarrow \{C_1, \ldots, C_n\}$ components
$popsize \leftarrow$ number of trails to build at once
$\alpha \leftarrow$ elitist learning rate
$\beta \leftarrow$ evaporation rate
$\gamma \leftarrow$ initial value for pheromones
$\delta \leftarrow$ tuning parameter for heuristics in component selection ▷ Usually $\delta = 1$
$\epsilon \leftarrow$ tuning parameter for pheromones in component selection
$t \leftarrow$ iterations to Hill-Climb
$q \leftarrow$ probability of selecting components in an elitist way

$p \leftarrow \langle \rangle$ pheromones of the components, all set to γ
$Best \leftarrow \square$
repeat
  $P \leftarrow \{\}$ ▷ Our candidate solutions
  for $popsize$ times **do**
    $S \leftarrow \{\}$
    repeat
      $C' \leftarrow$ components in $C - S$ which could be added to S without being infeasible
      if C' is empty **then** ▷ Try again
        $S \leftarrow \{\}$
      else
        $S \leftarrow S \cup \{$component selected from C' based on pheromones and values or costs$\}$
    until S is a complete trail
    $S \leftarrow$ Hill-Climb(S) for t iterations ▷ Optinal. By default, not done.
    AssessFitness(S)
    if $Best = \square$ or Fitness$(S) >$ Fitness$(Best)$ **then**
      $Best \leftarrow S$
  for each $p_i \in p$ **do** ▷ Decrease all pheromones a bit ("evaporation")
    $p_i \leftarrow (1 - \beta)p_i + \beta\gamma$
  for each $P_j \in P$ ▷ Update pheromones in components used in trails
    for each component S_i **do**
      if S_i was used in $Best$ **then** ▷ Update pheromones only of components in Best
        $p_i \leftarrow (1 - \alpha)p_i + \alpha$Fitness$(Best)$
until $Best$ is the ideal solution or we have run out of time
return $Best$

- the selection of components in candidate solutions is greedily based on how well a component has appeared in high-quality solutions (or perhaps even the best solution so far)

- it doesn't consider the possibility that a component needs to always appear with some other component in order to be good, and without the second component it's terrible

- ACO has a lot in common with Univariate Estimation of Distribution Algorithms (like CMAES we have seen last year)

- the components' fitnesses may be viewed as probabilities and the whole population is thus one probability distribution on a per-component basis

- contrast this to the evolutionary model, where the population may also be viewed as a sample distribution over the joint space of all possible candidate solutions, that is, all possible combinations of components

- it should be obvious that ACO is searching a radically simpler (perhaps simplistic) space compared to the evolutionary model

- for general problems that may be an issue, but for many combinatorial problems, it's proven to be a good tradeoff

Policy Optimization

7 Approximate Dynamic Programming

- we have seen that it is sometimes possible to use the DP algorithm to obtain an optimal policy in closed form (e.g. in LQR)

- however, this tends to be the exception - in most cases a numerical solution is necessary

- the associated computational requirements are of the overwhelming, and for many problems a complete solution of the problem by DP is impossible

- to a great extent, the reason lies in what R. Bellman has called the "curse of dimensionality" - an exponential increase of the required computation as the problem's size increases

- consider of example a problem where the state, control, and disturbance spaces are the Euclidean spaces $\mathbb{R}^n, \mathbb{R}^m$, and $\mathbb{R}^r$, respectively:

  - in a straightforward numerical approach, these spaces can be discretized - taking d discretization points per state axis results in a state space grid of d^n points

  - for each of these points the minimization of the RHS of the DP equation must be carried out numerically, which involves comparison of as many as d^m numbers

  - to calculate each of those numbers, one must calculate an expected value over the disturbance, which is a weighted sum of as many as d^r numbers

  - finally, the calculation must be done for each of the N stages

  - thus as a first approximation, the number of computational operations is at least of the order of $N d^n$ and can be of the order of $N d^{n+m+r}$

- it follows that for perfect state information problems with Euclidean state and control spaces, DP can be applied numerically only if the dimensions of the spaces are relatively small

- also, based on the analysis of the chapter on imperfect information, we can also conclude that for problems of imperfect state information the situation is hopeless, except for very simple or very special cases

- in the real world, there is an additional aspect of optimal control problems that can have a profound impact on the feasibility of DP as a practical solution method - in particular, there are many circumstances where the structure of the given problem is known well in advance, but some of the problem data, such as various system parameters, may be unknown until shortly before control is needed, thus seriously constraining the amount of time available for the DP computation

- Usually this occurs as a result of one or both of the following situations:

(a) **a family of problems is addressed, rather than a single problem**, and we do not get to know the exact problem to be solved until shortly before the control process begin

  - as an example, consider a problem of planning the daily route of a utility vehicle within a street network so that it passes through a number of points where it must perform some service - the street network and the vehicle characteristics may be known well in advance, but the service points may vary from day to day, and may not become known until shortly before the vehicle begins its route

  - this example is typical of situations, where the same problem must periodically be solved with small variations in its data - yet, if DP is to be used, the solution of one instance of the problem may not help appreciably in solving a different instance

(b) **the problem data changes as the system is being controlled**

- as an example, consider the route planning example in case (a) above, and assume that new service points to be visited arise as the vehicle is on its way

- it is possible in principle to model these data changes in terms of stochastic disturbances, but then we may end up with a problem that is too complicated for analysis or solution by DP

- a frequently employed alternative is to use **on-line replanning**, whereby the problem is resolved on-line with the new data, as soon as these data become available, and control continues with a policy that corresponds to the new data

- a common feature of the above situations, which can seriously impact the solution, is that there may be stringent time constraints for the computation of the controls - this may substantially exacerbate the "curse of dimensionality" problem mentioned above

Certainty Equivalent and Adaptive Control

- the **certainty equivalent controller** (CEC) is a suboptimal control scheme that is inspired by linear-quadratic control theory

- it applies at each stage the control that would be optimal if the uncertain quantities were fixed at some "typical" values; that is, it acts as if a form of the certainty equivalence principle were holding

- the advantage of the CEC is that it replaces the DP algorithm with what is often a much less demanding computation: the solution of a **deterministic** optimal control problem at each stage

- this problem yields an optimal control sequence, the first component of which is used at the current stage, while the remaining components are discarded

- the main attractive characteristic of the CEC is its ability to deal with stochastic and even imperfect information problems by using the mature and effective methodology of deterministic optimal control

- we describe the CEC for the general problem with imperfect state information - as can be expected, the implementation is considerably simpler if the controller has perfect state information

- suppose that we have an "estimator" that uses the information vector I_k to produce a "typical" value $\bar{x}_k(I_k)$ of the state, and also assume that for every state-control pair (x_k, u_k) we have selected a "typical" value of the disturbance, which we denote by $\bar{w}_k(x_k, u_k)$

- for example, if the state spaces and disturbance spaces are convex subsets of Euclidean spaces, the expected values

$$\bar{x}_k(I_k) = E\{x_k \mid I_k\}, \quad \bar{w}_k(x_k, u_k) = E\{w_k \mid x_k, u_k\}$$

can serve as typical values

- the control input $\bar{\mu}_k(I_k)$ applied by the CEC at each time k is determined by the following rule:

(1) Given the information vector I_k compute the state estimate $\bar{x}_k(I_k)$

(2) Find a control sequence $\{\bar{u}_k, \bar{u}_{k+1}, \ldots, \bar{u}_{N-1}\}$ that solves the deterministic problem obtained by fixing the uncertain quantities x_k and $w_k, \ldots, w_{N-1}$ at their typical values:

$$\min \ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, u_i, \bar{w}_i(x_i, u_i))$$

subject to the initial condition $x_k = x_k(I_k)$ and the constraints

$$u_i \in U_i, \quad x_{k+1} = f_i(x_i, u_i, \bar{w}_i(x_i, u_i)), \quad i = k, k+1, \ldots, N-1$$

(3) Use as control the first element in the control sequence found

$$\hat{\mu}_k(I_k) = \bar{u}_k$$

351
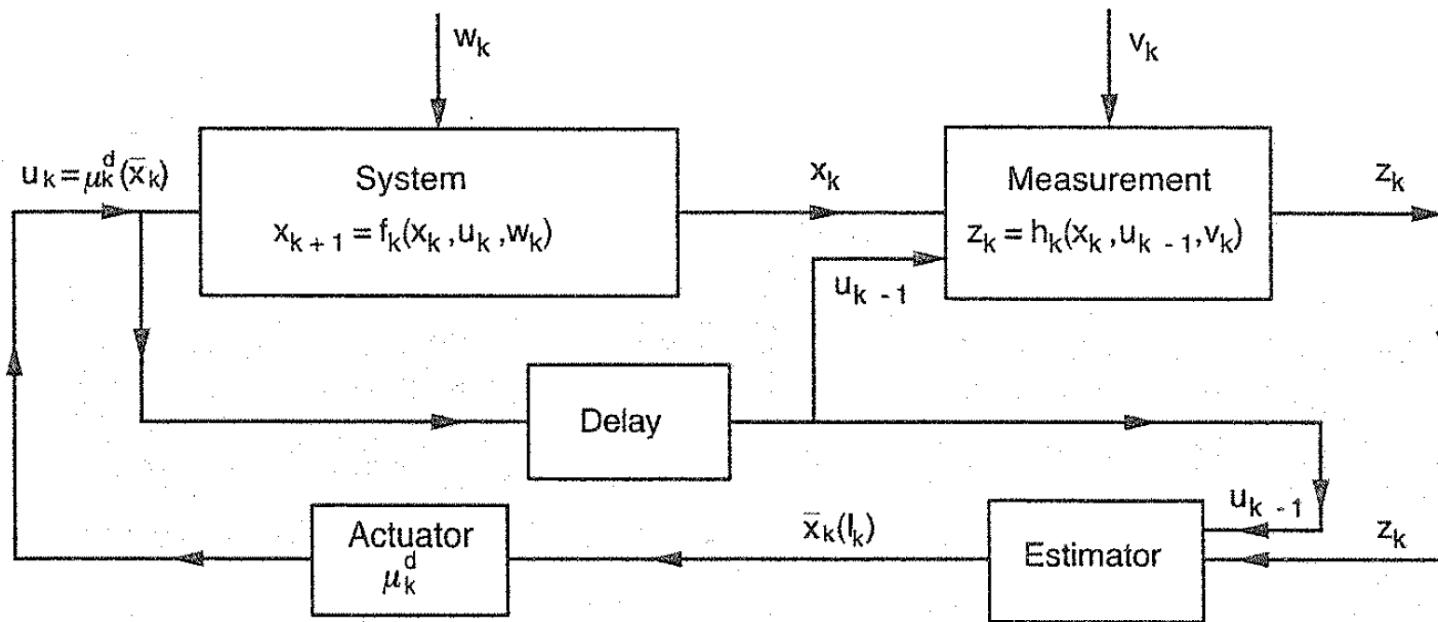
- note that step (1) is unnecessary if we have perfect state information; in this case we simply use the known value of the x_k

- the deterministic optimization problem in step (2) must be solved at each time k, once the initial state $\bar{x}_k(I_k)$ becomes known by means of an estimation (or perfect observation) procedure

- a total of N such problems must be solved by the CEC at every system run - in many cases of interest, these deterministic problems can be solved by powerful numerical methods such as conjugate gradient, Newton's method, augmented Lagrangian, and sequential quadratic programming methods (that we have seen in the previous course)

- furthermore, the implementation of the CEC requires no storage of the type required for the optimal feedback controller

- an alternative to solving N optimal control problems in an "on-line" fashion is to solve these problems a priori

- this is accomplished by computing an optimal feedback controller for the deterministic optimal control problem obtained from the original problem by replacing all uncertain quantities by their typical values

- it can be verified, that the implementation of the CEC given earlier is equivalent to the following

- let $\{\mu_0^d(x_0), \ldots, \mu_{N-1}^d(x_{N-1})\}$ be an optimal controller obtained from the DP algorithm for the deterministic problem

$$\min g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, \bar{w}_k(x_k, u_k))$$

$$\text{s.t. } x_{k+1} = f_i(x_k, u_k, \bar{w}_k(x_k, u_k)), \quad \mu_k(x_k) \in U_k, \quad k \geq 0,$$

then the control input $\bar{\mu}_k(I_k) = \mu_k^d(\bar{x}_k(I_k))$ as shown on the figure on the following slide

w_k

v_k

$u_k = \mu_k^d(\bar{x}_k)$

| System $x_{k+1} = f_k(x_k, u_k, w_k)$ | x_k | Measurement $z_k = h_k(x_k, u_{k-1}, v_k)$ | z_k |

u_{k-1}

Delay

Actuator μ_k^d

$\bar{x}_k(I_k)$

u_{k-1}

z_k

Estimator

- an equivalent alternative implementation of the CEC consists of finding a feedback controller $\{\mu_0^d, \ldots, \mu_{N-1}^d\}$ that is optimal for a corresponding deterministic problem, and subsequently using this controller for control of the uncertain system [modulo substitution of the state x_k by its estimate $\bar{x}_k(I_k)$]; either one of the definitions given for the CEC can serve as a basis for its implementation

- the CEC approach often performs well in practice and yields near-optimal policies (and is optimal in the LQR setting)

- even though the CEC approach simplifies a great deal the computations, it still requires the solution of a deterministic optimal control problem at each stage - this problem may be difficult, and a more convenient approach may be to solve it **suboptimally using a heuristic algorithm**

- let us assume the perfect information setting (for imperfect setting, substitute x_k with $\bar{x}_k(I_k)$); in this approach, given x_k, we use some (easily implementable) heuristic to find a suboptimal control sequence $\{\bar{u}_k, \bar{u}_{k+1}, \ldots, \bar{u}_{N-1}\}$ for the problem

$$\min g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, u_i, \bar{w}_i(x_i, u_i))$$

$$\text{s.t. } x_{i+1} = f_i(x_i, u_i, \bar{w}_i(x_i, u_i)), \quad u_i \in U_i(x_i), \quad i = k, k+1, \ldots, N-1$$

and we then use $\bar{u}_k$ as the control for stage k

- an important enhancement of this idea is to use minimization over the first control u_k and to use the heuristic only for the remaining stages

- to implement this variant of the CEC, we must apply at time k a control $\bar{u}_k$ that minimizes over $u_k \in U_k(x_k)$ the expression

$$g_k(x_k, u_k, \bar{w}_k(x_k, u_k)) + H_{k+1}(f_k(x_k, u_k, \bar{w}_k(x_k, u_k)))$$

where H_{k+i} is the cost-to-go function corresponding to the heuristic, i.e., $H_{k+i}(x_{k+1})$ is the cost incurred over the remaining stages $k+1, \ldots, N-1$ starting from a state x_{k+1}, using the heuristic, and assuming that the future disturbances will be equal to their typical values $\bar{w}_i(x_i, u_i)$

- it is not needed to have a closed-form expression for the heuristic cost-to-go $H_{k+1}(x_{k+1})$ - we can generate this cost by running the system forward from x_{k+1} and accumulating the corresponding single-stage costs

- since the heuristic must be run for each possible value of the control u_k to calculate the costs $H_{k+1}(f_k(x_k, u_k, \bar{w}_k(x_k, u_k)))$ needed in the minimization, it is necessary to discretize the control constraint set if it is not already finite

- note that the general structure of the preceding variant of the CEC is similar to the one of standard DP - it involves minimization of an expression, which is the sum of a current stage cost and a cost-to-go starting from the next state

- the difference with DP is that the optimal cost-to-go $J^*_{k+1}(x_{k+1})$ is replaced by the heuristic cost $H_{k+1}(x_{k+1})$, and the disturbance w_k is replaced by its typical value $\bar{w}_k(x_k, u_k)$ (so that there is no need to take expectation over w_k)

- we thus encounter for the first time an important suboptimal control idea, based on an approximation to the DP algorithm: *minimizing at each stage k the sum of approximations to the current stage cost and the optimal cost-to-go*

- this idea is central in other types of suboptimal control such as the limited lookahead, rollout, and model predictive control approaches, which will be discussed further

Sufficient Statistics

- the main difficulty with the DP algorithm for imperfect state information problems is that it is carried out over a state space of expanding dimension - as a new measurement is added at each stage k, the dimension of the state (the information vector I_k) increases accordingly

- this motivates an effort to reduce the data that are truly necessary for control purposes - in other words, it is of interest to look for quantities known as **sufficient statistics**, which ideally would be of a smaller dimension than I_k and yet summarize all the essential content of I_k as far as control is concerned

- consider the DP algorithm for the imperfect state information problem:

$$J_{N-1}(I_{N-1}) = \min_{u_{N-1} \in U_{N-1}} \Big[E_{x_{N-1}, w_{N-1}} \big\{ g_N(f_{N-1}(x_{N-1}, u_{N-1}, w_{N-1}))$$

$$+ g_{N-1}(x_{N-1}, u_{N-1}, w_{N-1}) \,|\, I_{N-1}, u_{N-1} \big\} \Big],$$

and for $k = 0, 1, \ldots, N-2$,

$$J_k(I_k) = \min_{u_k \in U_k} \Big[E_{x_k, w_k, z_{k+1}} \big\{ g_k(x_k, u_k, w_k) + J_{k+1}(I_k, z_{k+1}, u_k) \,|\, I_k, u_k \big\} \Big]$$

- suppose that we can find a function $S_k(I_k)$ of the information vector, such that minimizing control in the above equations depends on I_k via $S_k(I_k)$, i.e. the minimization of the RHS can be written in terms of some function H_k as

$$\min_{u_k \in U_k} H_k(S_k(I_k), u_k)$$

- such a function S_k is called a **sufficient statistic** - its salient feature is that an optimal policy obtained by the minimization can be written as

$$\mu_k^*(I_k) = \bar{\mu}_k(S_k(I_k)),$$

where $\bar{\mu}_k$ is an appropriate function

- thus, if the sufficient statistic is characterized by a set of fewer numbers than the information vector I_k, it may be easier to implement the policy in the form $u_k = \bar{\mu}_k(S_k(I_k))$ and take advantage of the resulting data reduction

- there are many different functions that can serve as sufficient statistic, for instance the identity function $S_k(I_k) = I_k$ is one of them (although it is not very useful)

- another important sufficient statistic is the conditional probability distribution $P_{x_k \mid I_k}$ of the state x_k, given the information vector I_k

- one extra assumption is necessary: **the probability distribution of the observation disturbance v_{k+1} depends explicitly only on the immediate preceding state, control, and system disturbance x_k, u_k, w_k and not on** $x_{k-1}, \ldots, x_0, u_{k-1}, \ldots, u_0, w_{k-1}, \ldots, w_0, v_{k-1}, \ldots, v_0$

- under this assumption, it can be shown [Bertsekas, p. 253-254] that for all k and I_k we have

$$J_k(I_k) = \min_{u_k \in U_k} H_k(P_{x_k \mid I_k}, u_k) = \bar{J}_k(P_{x_k \mid I_k}),$$

  where H_k and $\bar{J}_k$ are appropriate functions

- if the conditional distribution $P_{x_k \mid I_k}$ is uniquely determined by another expression $S_k(I_k)$, i.e.,

$$P_{x_k \mid I_k} = G_k(S_k(I_k))$$

  for an appropriate function G_k, then $S_k(I_k)$ is also sufficient statistic

- thus, for example, if we can show that $P_{x_k \mid I_k}$ is a Gaussian distribution, then the mean and the covariance matrix corresponding to $P_{x_k \mid I_k}$ form a sufficient statistic

- regardless of its computational value, the representation of the optimal policy as a sequence of functions on the conditional probability distribution $P_{x_k \mid I_k}$,

$$\mu_k(I_k) = \bar{\mu}_k(P_{x_k \mid I_k}), \quad k = 0, 1, \ldots, N - 1,$$

  is conceptually very useful

- it provides a decomposition of the optimal controller in two parts:

  (a) an **estimator**, which uses at time k the measurement z_k and the control u_{k-1} to generate probability distribution $P_{x_k \mid I_k}$

  (b) an **actuator**, which generates a control input to the system as a function of the probability distribution $P_{x_k \mid I_k}$

- this interpretation has formed the basis of various suboptimal control schemes that separate the controller a priori into an estimator and an actuator and attempt to design each part in a manner that seems "reasonable"

Open-Loop Feedback Control

- generally, in a problem with imperfect state information, the performance of the optimal policy improves when extra information is available

- however, the use of this information may make the DP calculation of the optimal policy intractable - this motivates an approximation, based on a more tractable computation that in part ignores the availability of extra information

- let us consider the imperfect state information problem with the assumption from the previous section, which guarantees that the conditional state distribution is a sufficient statistic, i.e., that the probability distribution of the observation disturbance v_{k+1} depends explicitly only on the immediately preceding state, control, and system disturbance x_k, u_k, w_k, and not $x_{k-1}, \ldots, x_0, u_{k-1}, \ldots, u_0, w_{k-1}, \ldots, w_0, v_{k-1}, \ldots, v_0$

- we introduce a suboptimal policy known as the **open-loop feedback con-troller** (OLFC), which uses the current information vector I_k to determine $P_{x_k \mid I_k}$

- however, it calculates the control u_k as if no further measurements will be received, by using an open-loop optimization over the future evolution of the system; in particular, u_k is determined as follows:

  (1) given the information vector I_k, compute the conditional probability distribution $P_{x_k \mid I_k}$ (in the case of perfect state information, where I_k includes x_k, this step is unnecessary)

  (2) find control sequence $\{\bar{u}_k, \bar{u}_{k+1}, \ldots, \bar{u}_{N-1}\}$ that solves the open-loop problem of minimizing

  $$E \left\{ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, u_i, w_i) \mid I_k \right\}$$

  subject to constraints

  $$x_{i+1} = f_i(x_i, u_i, w_i), \quad u_i \in U_i, \quad i = k, k+1, \ldots, N-1$$

(3) apply to control input

$$\bar{\mu}_k(I_k) = \bar{u}_k$$

- thus the OLFC uses at time k the new measurement z_k to calculate the conditional probability distribution $P_{x_k \mid I_k}$ However, it selects the control input as if future measurements will be disregarded

- similar to the CEC, the OLFC requires the solution of N optimal control problems - each problem may again be solved by deterministic optimal control techniques

- the computations, however, may be more complicated than those for the CEC, since now the cost involves an expectation with respect to the uncertain quantities

- the main difficulty in the implementation of the OLFC is the computation of $P_{x_k \mid I_k}$ - in many cases one cannot compute $P_{x_k \mid I_k}$ exactly, in which case some "reasonable" approximation scheme must be used (of course, if we have perfect state information, this difficulty does not arise)

- in any suboptimal control scheme, one would like to be assured that measurements are advantageously used, i.e., that the scheme performs at least as well as any open-loop policy that uses a sequence of controls that is independent of the values of the measurements received

- an optimal open-loop policy can be obtained by finding a sequence $\{u_0^*, u_1^*, \ldots, u_{N-1}^*\}$ that minimizes

$$\bar{J}(u_0, u_1, \ldots, u_{N-1}) = E\left\{g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k)\right\}$$

subject to constraints

$$x_{k+1} = f_k(x_k, u_k, w_k), \quad u_k \in U_k, \quad k = 0, 1, \ldots, N-1$$

- a nice property of OLFC is that it performs at least as well as an optimal open-loop policy (shown in the book, p. 301); but the CEC does not have this property (for a one-stage problem, the optimal open-loop policy and the OLFC are both optimal, but the CEC may be strictly suboptimal)

Limited Lookahead Policies

- an effective way to reduce the computation required by DP is to truncate the time horizon and use at each stage a decision based on lookahead of a small number of stages

- the simplest possibility is to use a **one-step lookahead policy** whereby at stage k and state x_k one uses the control $\bar{\mu}_k(x_k)$, which attains the minimum in the expression

$$\min_{u_k \in U_k(x_k)} E\left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right\}$$

where $\tilde{J}_{k+1}$ is some approximation of the true cost-to-go function J_{k+1} with $\tilde{J}_N = g_N$

- similarly, a **two-step lookahead policy** applies at time k and state x_k, the control $\bar{\mu}_k(x_k)$ attaining the minimum in the preceding equation, where now $\tilde{J}_{k+1}$ is obtained itself on the basis of a one-step lookahead approximation, i.e., for all possible states x_{k+1} that can be generated via the system equation starting from x_k

$$x_{k+1} = f_k(x_k, u_k, w_k)$$

we have

$$\tilde{J}_{k+1}(x_{k+1}) = \min_{u_{k+1} \in U_{k+1}(x_{k+1})} E\left\{g_{k+1}(x_{k+1}, u_{k+1}, w_{k+1}) + \tilde{J}_{k+2}(f_{k+1}(x_{k+1}, u_{k+1}, w_{k+1}))\right\}$$

where $\tilde{J}_{k+2}$ is some approximation of the true cost-to-go function J_{k+2}

- policies with lookahead of more than two stages are similarly defined
- note that the limited lookahead approach can be used equally well when the horizon is infinite - one simply uses as the terminal cost-to-go function an approximation to the optimal cost of the infinite horizon problem that starts at the end of the lookahead

- given the approximations $\tilde{J}_k$ to the optimal costs-to-go, the computational savings of the limited lookahead approach are evident

- for a one-step lookahead policy, only a single minimization problem has to be solved per stage, while in a two-step policy the corresponding number of minimization problems is one plus the number of all possible next states x_{k+1} that can be generated from the current state x_k

- however, even with readily available cost-to-go approximations $\tilde{J}_k$, the minimization over $u_k \in U_k(x_k)$ in the calculation of the one-step lookahead control may involve substantial computation.

- in a variant of the method that aims at reducing this computation, the minimization is done over a subset

$$\bar{U}_k(x_k) \subset U_k(x_k)$$

- thus, the control $\bar{\mu}_k(x_k)$ used in this variant is one that attains the minimum in the expression

$$\min_{u_k \in \bar{U}_k(x_k)} E\left\{g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k))\right\}$$

- a practical example of this approach is when by using some heuristic or approximate optimization, we identify a subset $\bar{U}_k(x_k)$ of promising controls, and to save computation, we restrict attention to this subset in the one-step lookahead minimization

- now, let us denote by $\bar{J}_k(x_k)$ the expected cost-to-go incurred by a limited lookahead policy $\{\bar{\mu}_0, \bar{\mu}_1, \ldots, \bar{\mu}_{N-1}\}$ starting from state x_k at time k

- it is generally difficult to evaluate analytically the functions $\bar{J}_k$, even when the functions $\tilde{J}_k$ are readily available; we thus aim to obtain some estimates of $\bar{J}_k(x_k)$.

- the following proposition gives a condition under which the one-step lookahead policy achieves a cost $\bar{J}_k(x_k)$ that is better than the approximation $\tilde{J}_k(x_k)$ and the proposition also provides a readily computable upper bound to $\bar{J}_k(x_k)$ [proof in the book, p. 306]

Proposition 5.1: Assume that for all x_k and k, we have

$$\min_{u_k \in \bar{U}_k(x_k)} E\left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right\} \leq \tilde{J}_k(x_k)$$

Then the cost-to-go functions $\bar{J}_k$ corresponding to a one-step lookahead policy that uses $\tilde{J}_k$ and $\bar{U}_k(x_k)$ satisfy for all x_k and k

$$\bar{J}_k(x_k) \leq \min_{u_k \in \bar{U}_k(x_k)} E\left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right\}$$

- we obtain for all x_k and k, that $\bar{J}_k(x_k) \leq \tilde{J}_k(x_k)$, i.e., **the cost-to-go of the one-step lookahead policy is no greater than the lookahead approximation on which it is based** (if the assumption of the proposition holds)

Example 7.1 – Rollout Algorithm

- suppose that $\tilde{J}_k(x_k)$ is the cost-to-go of some given (suboptimal) heuristic policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ and that the set $\bar{U}_k(x_k)$ contains the control $\mu_k(x_k)$ for all x_k and k

- the resulting one-step lookahead algorithm is called the **rollout algorithm** and will be discussed extensively in further sections; from the DP algorithm (restricted to the given policy π), we have

$$\tilde{J}_k = E\left\{ g_k(x_k, \mu_k(x_k), w_k) + \tilde{J}_{k+1}(f_k(x_k, \mu_k, w_k)) \right\}$$

which in view of the assumption $\mu_k(x_k) \in \bar{U}_k(x_k)$, yields

$$\tilde{J}_k \geq \min_{u_k \in \bar{U}_k(x_k)} E\left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right\}$$

- thus, the assumption of Prop. 5.1 is satisfied, and it follows that the rollout algorithm performs better than the heuristic on which it is based, starting from any state and stage

- we now discuss the computation of the cost-to-go approximations and the corresponding minimization of the one-step lookahead costs

- one approach to obtain the control $\bar{\mu}_k(x_k)$ used by the one-step lookahead policy is to exhaustively calculate and compare the one-step lookahead costs of all the controls in the set $\bar{U}_k(x_k)$

- in some cases, there is a more efficient alternative, which is to solve a suitable nonlinear programming problem

- in particular, if the control space is the Euclidean space $\mathbb{R}^m$, then for a one-step lookahead control calculation, we are faced with a minimization over a subset of $\mathbb{R}^m$, which may be approached by continuous optimization/nonlinear programming techniques

- it turns out that even a multistage lookahead control calculation can be approached by nonlinear programming

- in particular, assume that the disturbance can take a finite number of values, say r; then, it can be shown that for a given initial state, an l-stage perfect state information problem (which corresponds to an l-step lookahead control calculation) can be formulated as a nonlinear programming problem with $m(1 + r^{l-1})$ variables

- we illustrate this by means of an important example where $l = 2$ and then discuss the general case

Example 7.2 – Two-Stage Stochastic Programming

- here we want to find an optimal two-stage decision rule for the following situation: in the first stage we will choose a vector u_o from a subset $U_o \subset \mathbb{R}^m$ with cost $g_o(u_o)$

- then an uncertain event represented by a random variable w will occur, where w will take one of the values $w^1, \ldots, w^r$ with corresponding probabilities $p^1, \ldots, p^r$

- we will know the value w^j once it occurs, and we must then choose a vector u_1^j from a subset $U_1(u_0, w^j)$ at a cost $g_1(u_1^j, w^j)$

- the objective is to minimize the expected cost

$$g_0(u_0) + \sum_{j=1}^{r} p^j g_1(u_1^j, w^j)$$

subject to

$$u_0 \in U_0, \quad u_1^j \in U_1(u_0, w^j), \quad j = 1, \ldots, r$$

- this is a nonlinear programming problem of dimension $m(1+r)$ (the optimization variables are $u_o, u_1^1, \ldots, u_1^r$)

- it can also be viewed as a two-stage perfect state information problem, where $x_1 = w_o$ is the state equation, w_o can take the values $w^1, \ldots, w^r$ with probabilities $p^1, \ldots, p^r$, the cost of the first stage is $g_0(u_0)$, and the cost of the second stage is $g_1(x_1, u_1)$

- the preceding example can be generalized - consider the basic problem of Chapter 1 for the case where there are only two stages ($l = 2$) and the disturbances w_0 and w_1 can independently take one of the r values $w^1, \ldots, w^r$ with corresponding probabilities $p^1, \ldots, p^r$

- the optimal cost

$$
J_0(x_0) = \min_{u_0 \in U_0(x_0)} \Bigg[\sum_{j=1}^{r} p^j \Big\{ g_0(x_0, u_0, w^j)
$$

$$
+ \min_{u_1^j \in U_1(f_0(x_0, u_0, w^j))} \Big[\sum_{i=1}^{r} p^i \big\{ g_1(f_0(x_0, u_0, w^j), u_1, w^i) + g_2(f_1(f_0(x_0, u_0, w^j), u_1^j, w^i)) \big\} \Big] \Bigg\}
$$

- the DP algorithm is equivalent to solving the nonlinear problem

$$
\text{minimize } \sum_{j=1}^{r} \Big\{ g_0(x_0, u_0, w^j) + \sum_{i=1}^{r} p^i \big\{ g_1(f_0(x_0, u_0, w^j), u_1, w^i)
$$

$$
+ g_2(f_1(f_0(x_0, u_0, w^j), u_1^j, w^i)) \big\} \Big\}
$$

$$
\text{subject to } u_0 \in U_0(x_0), \quad u_1^j \in U_1(f_0(x_0, u_0, w^j)), \quad j = 1, \ldots, r
$$

- if the controls u_0 and u_1 are elements of $\mathbb{R}^m$ the number of variables in the above problem is $m(1 + r)$

- more generally, for an l-stage perfect state information problem a similar reformulation as a nonlinear programming problem requires $m(1 + r^{l-1})$ variables

- thus if the number of lookahead stages is relatively small, a nonlinear programming approach may be the preferred option in calculating sub-optimal limited lookahead policies

Choosing the Approximate Cost-to-Go

- a key issue in implementing a limited lookahead policy is the selection of the cost-to-go approximation at the final step

- it may appear important at first that the true cost-to-go function should be approximated well over the range of relevant states; however, this is not necessarily true

- what is important is that the cost-to-go differentials (or relative values) be approximated well; that is, for an l-step lookahead policy it is important to have

$$\tilde{J}_{k+l}(x) - \tilde{J}_{k+l}(x') \approx J_{k+l}(x) - J_{k+l}(x')$$

  for any two states x and x' that can be generated l steps ahead from the current state

- for example, if equality were to hold above for all x, x', then $\tilde{J}_{k+l}(x)$ and $J_{k+l}(x)$ would differ by the same constant for each relevant x and the l-step lookahead policy would be optimal

- the manner in which the cost-to-go approximation is selected depends very much on the problem being solved (with a wide variety of possibilities); three such approaches are:

  (a) **Problem Approximation**: The idea here is to approximate the optimal cost-to-go with some cost derived from a related but simpler problem (for example the optimal cost-to-go of that problem).

  (b) **Parametric Cost-to-Go Approximation**: The idea here is to approximate the optimal cost-to-go with a function of a suitable parametric form, whose parameters are tuned by some heuristic or systematic scheme.

  (c) **Rollout Approach**: Here the optimal cost-to-go is approximated by the cost of some suboptimal policy, which is calculated either analytically, or more commonly, by simulation. Generally, if a reasonably good suboptimal policy is known, it can be used to obtain a cost-to-go approximation. This approach is also particularly well-suited for deterministic and combinatorial problems.

Example 7.3 – Approximation Using Scenarios

- an often convenient approach for cost-to-go approximation is based on solution of a simpler problem that is tractable computationally or analytically

- one possibility to approximate the optimal cost-to-go is to use certainty equivalence (similar to the CEC control scheme)

- in particular, for a given state x_{k+1} at time $k+1$ we fix the remaining disturbances at some nominal values $\bar{w}_{k+1}, \ldots, \bar{w}_{N-1}$ and we compute an optimal control trajectory starting from x_{k+1} at time $k+1$

- the corresponding cost, denoted by $\tilde{J}_{k+1}(x_{k+1})$, is used to approximate the optimal cost-to-go $J_{k+1}(x_{k+1})$ for the purpose of computing the corresponding one-step lookahead policy

- thus to compute the one-step lookahead control at state x_k, we need to solve a deterministic optimal control problem from all possible next states $f_k(x_k, u_k, w_k)$ and to evaluate the corresponding optimal cost $\tilde{J}_{k+1}(f_k(x_k, u_k, W_k))$ based on the nominal values of the uncertainty

- a simpler but less effective variant of this approach is to compute $J_{k+1}(x_{k+1})$ as the cost-to-go of a given heuristic (rather than optimal) policy for the deterministic problem that corresponds to the nominal values of the uncertainty and the starting state x_{k+1}

- the advantage of using certainty equivalence here is that the potentially costly calculation of the expected value of the cost is replaced by a single state-control trajectory calculation

- the certainty equivalent approximation involves a single nominal trajectory of the remaining uncertainty

- to strengthen this approach, it is natural to consider multiple trajectories of the uncertainty, called **scenarios**, and to construct an approximation to the optimal cost-to-go that involves, for every one of the scenarios, the cost of either an optimal or a given heuristic policy

- mathematically, we assume that we have a method, which at a given state x_{k+1}, generates M uncertainty sequences

$$w^m(x_{k+1}) = (w_{k+1}^m, \ldots, w_{N-1}^m), \quad m = 1, \ldots, M$$

- these are the scenarios considered at state x_{k+1}, and the cost $J_{k+1}(x_{k+1})$ is approximated by

$$\tilde{J}_{k+1}(x_{k+1}, r) = r_0 + \sum_{m=1}^{M} r_m C_m(x_{k+1})$$

where $r = (r_0, r_1, \ldots, r_M)$ is a vector of parameters, and $C_m(x_{k+1})$ is the cost corresponding to an occurrence of the scenario $w^m(x_{k+1})$, when starting from state x_{k+1} and using either an optimal or a given heuristic policy

- the parameters $(r_0, r_1, \ldots, r_M)$ may depend on the time index, and in more sophisticated schemes, they may depend on some characteristics of the state

- we may interpret the parameter r_m, as an "aggregate weight" that encodes the aggregate effect on the cost-to-go function of uncertainty sequences that are similar to the scenario $w^m(x_{k+1})$

- note that, if $r_0 = 0$, the approximation may also be viewed as a calculation by limited simulation, based on just the M scenarios $w^m(x_{k+1})$, and using the weights r_m as "aggregate probabilities"

Aggregation

- an alternative method for constructing a simpler and more tractable problem is based on reducing the number of states by "combining" many of them together into **aggregate states**

- this results in an **aggregate problem**, with fewer states, which may be solvable by exact DP methods; the optimal cost-to-go functions of the aggregate problem is then used to construct a one-step-lookahead cost approximation for the original problem

- the precise form of the aggregate problem may depend on intuition and/or heuristic reasoning, based on our understanding of the original problem

- in this subsection, we will discuss various aggregation methods, starting with the case of a finite-state problem

- we will focus on defining the transition probabilities and costs of the aggregate problem, and to simplify notation, we suppress the time indexing in what follows:

$I, \bar{I}$: set of states of the original system at the current and the next stage

$p_{ij}(u)$: transition probability of the original system from state $i \in I$ to state $j \in \bar{I}$ under control u

$g(I, u, j)$: transition cost of the original system from state $i \in I$ to state $j \in \bar{I}$ under control u

$S, \bar{S}$: set of states of the aggregate system at the current and the next stage

$r_{st}(u)$: transition probability of the aggregate system from state $s \in S$ to state $t \in \bar{S}$ under control u

$h(s, u)$: expected transition cost of the aggregate system from state $s \in S$ under control u

- we also assume that the control constraint set $U(i)$ is the same for all states $i \in I$; this common control constraint set, denoted by U, is chosen as the control constraint set at all states $s \in S$ of the aggregate problem

- there are several types of aggregation methods, which bring to bear intuition about the problem's structure in different ways

- all these methods are based on two (somewhat arbitrary) choices of probabilities, which relate the original system states with the aggregate states:

  (1) for each aggregate state $s \in S$ and original system state $i \in I$, we specify the disaggregation probability q_{si} (we have $\sum_{i \in I} q_{si} = 1$ for each $s \in S$); roughly, q_{si} may be interpreted as the "degree to which s is represented by i"

  (2) for each original system state $j \in \bar{I}$ and aggregate state $t \in \bar{S}$, we specify the aggregation probability w_{jt} (we have $\sum_{t \in \bar{S}} w_{jt} = 1$ for each $j \in \bar{I}$); roughly, w_{jt} may be interpreted as the "degree of membership of j in the aggregate state t"

Example 7.4 – Hard Aggregation

- we are given a partition of the original system state spaces I and $\bar{I}$ into subsets of states (each state belongs to one and only one subset)

- we view each subset as an aggregate state - this corresponds to aggregation probabilities

$$w_{jt} = 1, \quad \text{if state } j \in \bar{I} \text{ belongs to aggregate state/subset } t \in \bar{S}$$

and (assuming all states that belong to aggregate state/subset s are "equally representative") disaggregation probabilities

$$q_{si} = 1/n_s, \quad \text{if state } i \in I \text{ belongs to aggregate state/subset } s \in S$$

where n_s is the number of states of s

- given the disaggregation and aggregation probabilities, q_{si} and w_{jt}, and the original transition probabilities $p_{ij}(u)$, we envisage an aggregate system where state transitions occur as follows:

  (i) from aggregate state s, generate state i according to q_{si}

  (ii) generate a transition from i to j according to $p_{ij}(u)$, with cost $g(i, u, j)$

  (iii) from state j, generate aggregate state t according to w_{jt}

- then, the transition probability from aggregate state s to aggregate state t under u, and the corresponding expected transition cost, are given by

$$r_{st}(u) = \sum_{i \in I} \sum_{j \in \bar{I}} q_{si} p_{ij}(u) w_{jt}, \quad h(s, u) = \sum_{i \in I} \sum_{j \in \bar{I}} q_{si} p_{ij}(u) g(i, u, j)$$
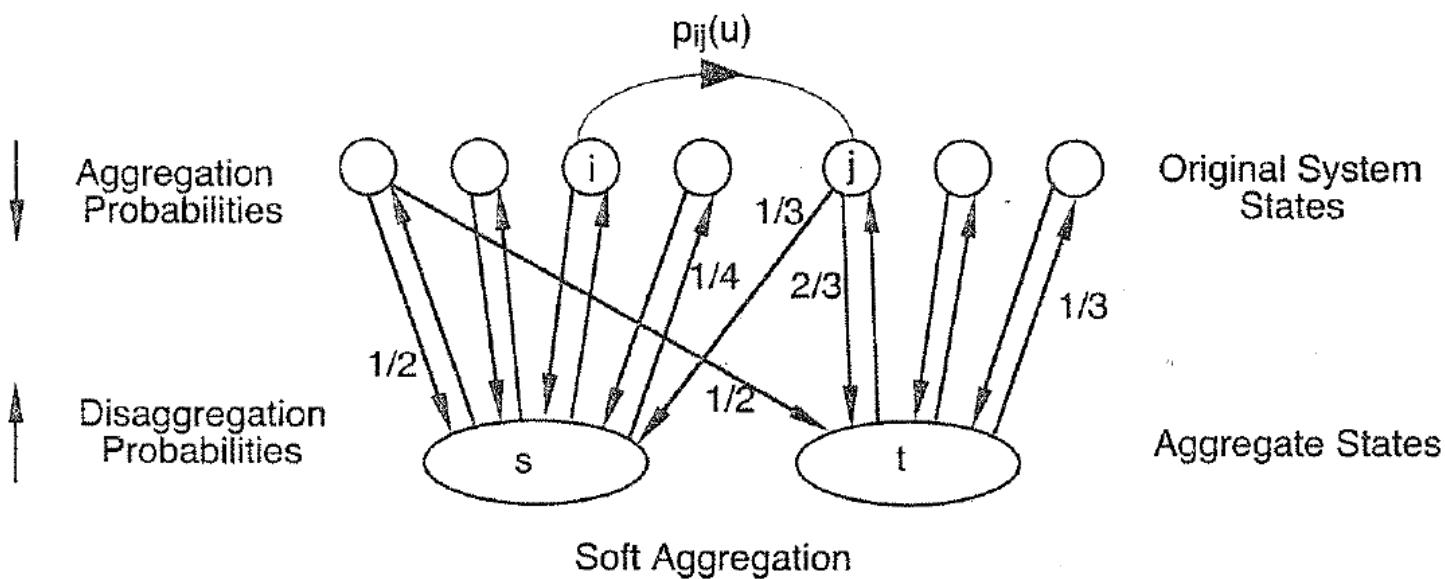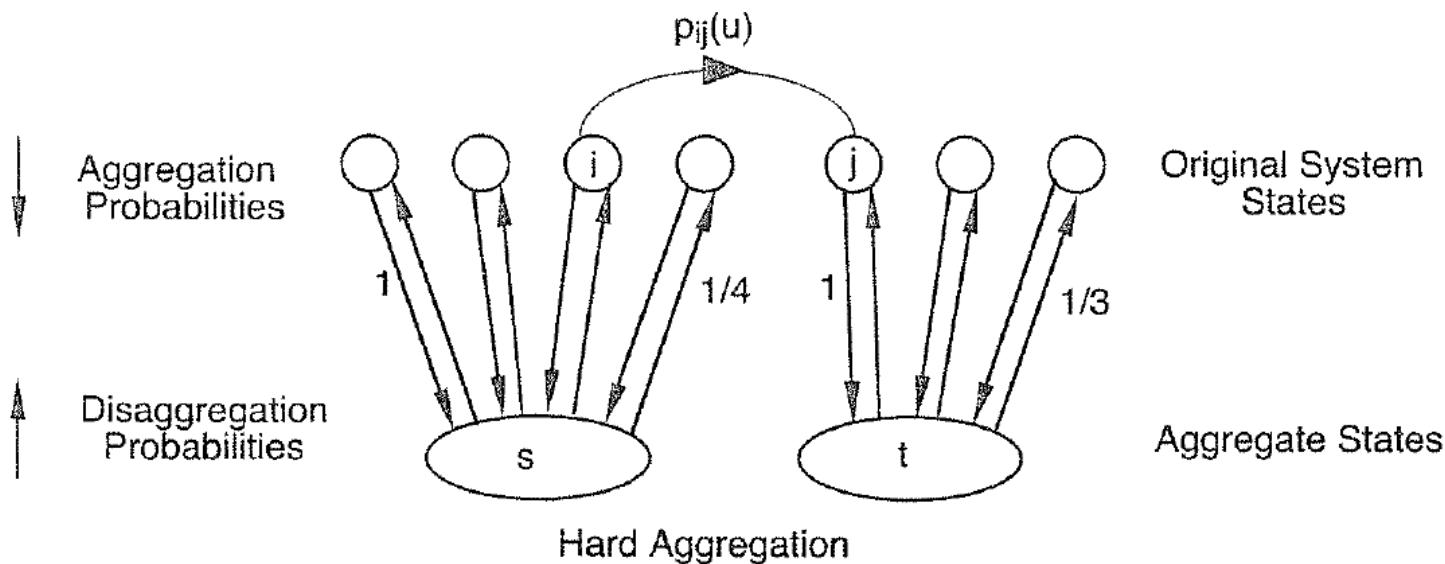
- These transition probabilities and costs define the aggregate problem - after solving for the optimal costs-to-go $\hat{J}(t), t \in \bar{S}$ of the aggregate problem, the costs of the original problem are approximated by

$$\tilde{J}(j) = \sum_{t \in \bar{S}} w_{jt} \hat{J}(t)$$

389

Example 7.5 – Soft Aggregation

- in hard aggregation, the aggregate states/subsets are disjoint, and each original system state is associated with a single aggregate state

- a generalization is to allow the aggregate states/subsets to overlap, with the aggregation probabilities w_{jt} quantifying the "degree of membership" of j in the aggregate state t

- thus, an original system state j may be a member of multiple aggregate states/subsets t, and if this is so, the aggregation probabilities w_{jt} will be positive but less than 1 for all t that contain j (we should still have $\sum_{t \in \bar{S}} w_{jt} = 1$)

- for example, assume that we are dealing with a queue that has space for 100 customers, and that the state is the number of spaces occupied at a given time

- suppose that we introduce four aggregate states: "nearly empty (0-10 customers), "lightly loaded" (11-50 customers), "heavily loaded" (51-90 customers), and "nearly full" (91-100 customers)

- then it makes sense to use soft aggregation, so that a state with close to 50 customers is classified neither as "lightly loaded" nor as "heavily loaded," but is viewed instead as associated with both of these aggregate states, to some degree

- in soft aggregation, the approximate cost-to-go function $\tilde{J}$ is not piecewise constant, as in the case of hard aggregation, and varies "smoothly" along the boundaries separating aggregate states; this is because original system states that belong to multiple aggregate states/subsets have approximate cost-to-go that is a convex combination of the costs-to-go of these aggregate states

$p_{ij}(u)$

Aggregation Probabilities

Original System States

1   1/4   1   1/3

Disaggregation Probabilities

s   t

Aggregate States

Hard Aggregation

$p_{ij}(u)$

Aggregation Probabilities

Original System States

1/3

1/4   2/3

1/3

1/2   1/2

Disaggregation Probabilities

s   t

Aggregate States

Soft Aggregation

Example 7.6 – Using a Coarse Grid

- a technique often used to reduce the computational requirements of DP is to select a small collection S of states from I and a small collection $\bar{S}$ of states from $\bar{I}$, and define an aggregate problem whose states are those in S and $\bar{S}$

- the aggregate problem is then solved by DP and its optimal costs are used to define approximate costs for all states in I and $\bar{I}$

- this process, is known as coarse grid approximation, and is motivated by the case where the original state spaces I and $\bar{I}$ are dense grids of points obtained by discretization of continuous state spaces, while the collections S and $\bar{S}$ represent coarser subgrids

- the aggregate problem may be formalized by specifying the disaggregation probabilities to associate the states of S with themselves (since $S \subset I$):

$$q_{ss} = 1, \quad q_{si} = 0 \text{ if } i \neq s, \quad s \in S$$

Example 7.7 – Discretization of Continuous State Spaces

- the aggregation methodology also applies to problems with an infinite number of states

- the only difference is that for each aggregate state, the disaggregation probabilities are replaced by a disaggregation distribution over the original system's state space

- among other possibilities, this type of aggregation provides methods for discretizing continuous state space problems

- as an example, assume for simplicity a stationary problem, where the state space of the original problem is a bounded region of a Euclidean space

- the idea here is to discretize this state space using some finite grid $\{x^1, \ldots, x^M\}$, and then to express each nongrid state by a linear interpolation of nearby grid states

- by this we mean that the grid states $x^1, \ldots, x^M$ are suitably selected within the state space, and each nongrid state x is expressed as

$$x = \sum_{m=1}^{M} w^m(x) x^m$$

  for some nonnegative weights $w^m(x)$, which add to 1 and are chosen on the basis of some geometric considerations

- we view the weights $w^m(x)$ as aggregation probabilities, and we specify the disaggregation probabilities to associate the grid states with themselves, i.e.,

$$q_{x^m x^m} = 1, \quad \text{for all } m$$

- The aggregation and disaggregation probabilities just given specify the aggregate problem, which has a finite state space, the set $\{x^1, \ldots, x^M\}$, and can be solved by DP to obtain the corresponding optimal costs-to-go $\hat{J}_k(x^m), m = 1, \ldots, M$ for each stage k

- then the cost-to-go of each nongrid state x at stage k is approximated by
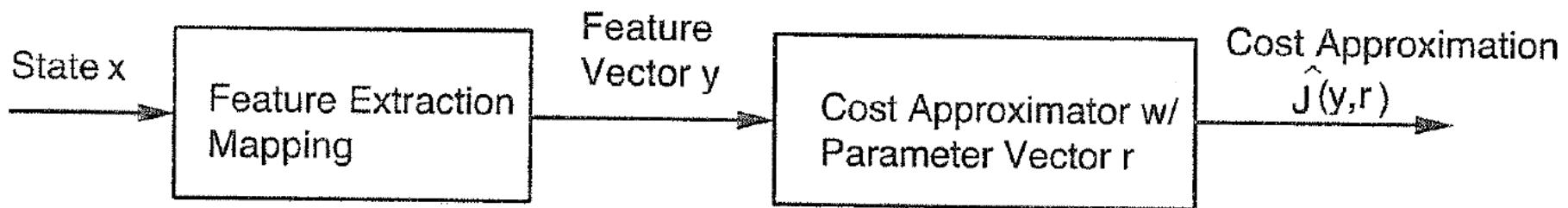
$$\tilde{J}_k(x_k) = \sum_{m=1}^{M} w^m(x)\hat{J}_k(x^m)$$

Parametric Cost-to-Go Approximation

- the idea here is to select from within a parametric class of functions, some cost-to-go approximations $\tilde{J}_k$, which will be used in a limited lookahead scheme in place of the optimal cost-to-go functions J_k

- such parametric classes of functions are called approximation architectures, and are generically denoted by $\tilde{J}(x, r)$, where x is the current state and $r = (r_1, \ldots, r_m)$ is a vector of "tunable" scalar parameters, also called weights (to simplify notation, we suppress the time indexing in what follows)

- by adjusting the weights, one can change the "shape" of the approximation $\tilde{J}$ so that it is reasonably close to the true optimal cost-to-go function

- there is an extensive methodology for the selection of the weights

- the simplest and often tried approach is to do some form of semi-exhaustive or semi-random search in the space of weight vectors and adopt the weights that result in best performance of the associated one-step lookahead controller

- other more systematic approaches are based on various forms of cost-to-go evaluation and least squares fit

- there is also a large variety of approximation architectures, involving for example polynomials, neural networks, wavelets, various types of basis functions, etc.

- we only briefly describe the architecture based on feature extraction

- clearly, for the success of the cost function approximation approach, it is very important to select a class of functions $\tilde{J}(x, r)$ that is suitable for the problem at hand

- one particularly interesting type of cost approximation is provided by feature extraction, a process that maps the state x into some other vector $y(x)$, called the feature vector associated with state x

- the vector $y(x)$ consists of scalar components $y_1(x), \ldots, y_m(x)$, called features

- these features are usually handcrafted, based on whatever human intelligence, insight, or experience is available, and are meant to capture the most important aspects of the current state x

- a feature-based cost approximation has the form

$$\tilde{J}(x, r) = \hat{J}(y(x), r),$$

  where r is a parameter vector

- thus, the cost approximation depends on the state x through its feature vector $y(x)$ (see figure above)

- the idea is that the cost-to-go function J to be approximated may be a highly complicated nonlinear map and it is sensible to try to break its complexity into smaller, less complex pieces

- ideally, the features will encode much of the nonlinearity that is inherent in J, and the approximation may be quite accurate without requiring a complicated function $\hat{J}$

- for example, with a well-chosen feature vector $y(x)$, a good approximation to the cost-to-go is often provided by linearly weighting the features, i.e.,

$$\tilde{J}(x,r) = \hat{J}(y(x), r) = \sum_{i=1}^{m} r_i y_i(x),$$

where $r_1, \ldots, r_m$ is a set of tunable scalar weights

- note that the use of a feature vector implicitly involves the grouping of states into the subsets that share the same feature vector, i.e., the subsets

$$S_v = \{x \mid y(x) = v\}$$

where v is possible value of $y(x)$

- these subsets form a partition of the state space, and the approximate cost-to-go function $\tilde{J}(y(x), r)$ is piecewise constant with respect to this partition; that is, it assigns the same cost-to-go value $\tilde{J}(v, r)$ to all states in the set S_v

- this suggests that a feature extraction mapping is well-chosen if states that have the same feature vector have roughly similar optimal cost-to-go

- a feature-based architecture is also related to the aggregation methodology discussed previously - in particular, suppose that we introduce m aggregate states, $1, \ldots, m$, and associated aggregation and disaggregation probabilities

- let r_i be the optimal cost-to-go associated with aggregate state i; then, the aggregation methodology yields the linear parametric approximation

$$\tilde{J}(x, r) = \sum_{i=1}^{m} r_i y_i(x)$$

where $y_i(x)$ is the aggregation probability associated with state x and aggregate state i

- thus, within this context, the aggregation probability $y_i(x)$ may be viewed as a feature, which, roughly speaking, specifies the "degree of membership of x to aggregate state i"

Rollout Algorithms

- we now discuss a specific type of cost-to-go approximation within the context of a limited lookahead scheme

- recall that in the one-step lookahead method, at stage k and state x_k we use the control $\bar{\mu}_k$ that attains the minimum in the expression

$$\min_{u_k \in U_k(x_k)} E\left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right\}$$

where $\tilde{J}_{k+1}$ is some approximation of the true cost-to-go function J_{k+1}

- in the rollout algorithm, the approximating function $\tilde{J}_{k+1}$ is the cost-to-go of some known heuristic/suboptimal policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ called **base policy**

- the policy thus obtained is called the **rollout policy** based on π

- thus **the rollout policy is a one-step lookahead policy, with the optimal cost-to-go approximated by the cost-to-go of the base policy**

- the process of starting from some suboptimal policy and generating another policy using the one-step lookahead process described above is also called **policy improvement**

- this process will also be discussed in an upcoming chapter in the context of the **policy iteration method**, which is a primary method for solving infinite horizon problems

- note that it is also possible to define rollout policies that make use of multistep (say, l-step) lookahead - here we assign to every state x that can be reached in l steps, the exact cost-to-go of the base policy, as computed by Monte Carlo simulation of several trajectories that start at x

- clearly, such multistep lookahead involves much more on-line computation, but it may yield better performance than its single-step counterpart; in what follows, we concentrate on rollout policies with single-step lookahead

- The viability of a rollout policy depends on how much time is available to choose the control following the transition to state x and on how expensive is the Monte Carlo evaluation of the expected value

$$E \left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right\}$$

- in particular, it must be possible to perform the Monte Carlo simulations and calculate the rollout control within the real-time constraints of the problem

- if the problem is deterministic, a single simulation trajectory suffices, and the calculations are greatly simplified, but in general, the computational overhead can be substantial

- it is possible, however, to speed up the calculation of the rollout policy if we are willing to accept some potential performance degradation; for example, we may use an approximation $\hat{J}_{k+1}$ of $\tilde{J}_{k+1}$ to identify a few promising controls through a minimization of the form

$$\min_{u_k \in U_k(x_k)} E\left\{ g_k(x_k, u_k, w_k) + \hat{J}_{k+1}(f_k(x_k, u_k, w_k)) \right\}$$

  and then restrict attention to these controls, using fairly accurate Monte Carlo simulation

- in particular, the required values of $\hat{J}_{k+1}$ may be obtained by performing approximately the Monte Carlo simulation, using a limited number of representative trajectories

- adaptive variants of this approach are also possible, whereby we use some heuristics to adjust the accuracy of the Monte Carlo simulation depending on the results of the computation

- generally, it is important to use as base policy one whose expected cost-to-go is conveniently calculated; the following is an example

Example 7.8 – The Quiz Problem

- consider the quiz problem we encountered previously, where a person is given a list of N questions and can answer these questions in any order he/she chooses

- question i will be answered correctly with probability p_i, and the person will then receive a reward v_i and at the first incorrect answer, the quiz terminates and the person is allowed to keep his or her previous rewards

- the problem is to choose the ordering of questions so as to maximize the total expected reward

- we saw that the optimal sequence can be obtained using an interchange argument: questions should be answered in decreasing order of the "index of preference" $p_i v_i / (1 - p_i)$; we refer to this as the index policy

- unfortunately, with only minor changes in the structure of the problem, the index policy need not be optimal; examples of difficult variations of the problem may involve one or more of the following characteristics:

a) A limit on the maximum number of questions that can be answered, which is smaller than the number of questions N. To see that the index policy is not optimal anymore, consider the case where there are two questions, only one of which may be answered. Then it is optimal to answer the question that offers the maximum expected reward $p_i v_i$.

b) A time window for each question, which constrains the set of time slots when each question may be answered. Time windows may also be combined with the option to refuse answering a question at a given period, when either no question is available during the period, or answering any one of the available questions involves excessive risk.

c) Precedence constraints, whereby the set of questions that can be answered in a given time slot depends on the immediately preceding question, and possibly on some earlier answered questions.

d) Sequence-dependent rewards, whereby the reward from answering correctly a given question depends on the immediately preceding question, and possibly on some questions answered earlier.

- nonetheless, even when the index policy is not optimal, it can conveniently be used as a base policy for the rollout algorithm

- the reason is that at a given state, the index policy together with its expected reward can be easily calculated; in particular, each feasible question order $(i_1, \ldots, i_N)$ has expected reward equal to

$$p_{i_1}(v_{i_1} + p_{i_2}(v_{i_2} + p_{i_3}(\cdots + p_{i_N} v_{i_N})\cdots))$$

- thus the rollout algorithm based on the index heuristic operates as follows:

  - at a state where a given subset of questions have already been answered, we consider the set of questions J that are eligible to be answered next

  - for each question $j \in J$, we consider a sequence of questions that starts with j and continues with the remaining questions chosen according to the index rule

  - we compute the expected reward of the sequence, denoted $R(j)$, using the above formula

  - then among the questions $j \in J$, we choose to answer next the one with maximal $R(j)$

- we now consider in more detail implementation issues and specific properties of rollout algorithms in a variety of settings

- to compute the rollout control $\bar{\mu}_k(x_k)$, we need for all $u_k \in U_k(x_k)$ the value of

$$Q_k(x_k, u_k) = E\left\{g_k(x_k, u_k, w_k) + H_{k+1}(f_k(x_k, u_k, w_k))\right\}$$

  known as the Q-**factor** of (x_k, u_k) at time k

- alternatively, for the computation of $\bar{\mu}_k(x_k)$ we need the value of the cost-to-go

$$H_{k+1}(f_k(x_u, u_k, w_k))$$

  of the base policy at all possible next states $f_k(x_k, u_k, w_k)$, from which we can compute the required Q-factors

- we will focus on the case where a closed form expression of the Q-factor is not available; we assume instead that we can simulate the system under the base policy π, and in particular, that we can generate sample system trajectories and corresponding costs consistently with the probabilistic data of the problem

- we will consider several cases and possibilities:

(1) **The deterministic problem** case, where wk takes a single known value at each stage. We provide an extensive discussion of this case, focusing not only on traditional deterministic optimal control problems, but also on quite general combinatorial optimization problems, for which the rollout approach has proved convenient and effective.

(2) **The stochastic problem case with Q-factors evaluated by Monte-Carlo simulation**. Here, once we are at state x_k, the Q-factors $Q_k(x_k, u_k)$ are evaluated on-line by Monte-Carlo simulation, for all $u_k \in U_k(x_k)$.

(3) **The stochastic problem case with Q-factors approximated** in some way. One possibility is to use a certainty equivalence approximation, where the problem is genuinely stochastic, but the values $H_k(x_k)$ are approximated by the cost-to-go of π that would be incurred if the system were replaced by a suitable deterministic system from state x_k and time k onward (assumed certainty equivalence).

- let us assume that the problem is deterministic, i.e., that w_k can take only one value at each stage k; then, starting from state x_k at stage k, the base policy π produces deterministic sequences of states $\{x_{k+1}, \ldots, x_N\}$ and controls $\{u_k, \ldots, u_{N-1}\}$ such that

$$x_{i+1} = f_i(x_i, u_i), \quad i = k, \ldots, N-1$$

  and costs

$$g_k(x_k, u_k) + \cdots + g_{N-1}(x_{N-1}, u_{N-1}) + g_N(x_N)$$

- thus, the Q-factor

$$Q_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k))$$

  can be obtained by running π starting from state $f_k(x_k, u_k)$ and time $k+1$, and recording the corresponding cost $H_{k+1}(f_k(x_k, u_k))$

- The rollout control $\bar{\mu}_k(x_k)$ is obtained by calculating in this manner the Q-factors $Q_k(x_k, u_k)$ for all $u_k \in U_k(x_k)$, and setting

$$\bar{\mu}_k(x_k) = \arg\min_{u_k \in U_k(x_k)} Q_k(x_k, u_k)$$

- aside for being convenient for the deterministic special case of the basic problem of Chapter 2, this rollout method can be adapted for general discrete or combinatorial optimization problems that do not necessarily have the strong sequential character of the basic problem

- for such problems the rollout approach provides a convenient and broadly applicable suboptimal solution method that goes beyond and indeed enhances the common types of heuristics, such as greedy algorithms, local search, genetic algorithms, tabu search, and others

- now, let us consider a stochastic problem and some computational issues regarding the implementation of the rollout policy based on a given heuristic policy

- a conceptually straightforward approach to compute the rollout control at a given state x_k and time k is to use Monte-Carlo simulation

- to implement this algorithm, we consider all possible controls $u_k \in U_k(x_k k)$ and we generate a "large" number of simulated trajectories of the system starting from x_k, using u_k as the first control, and using the policy π there-after

- thus a simulated trajectory has the form

$$x_{i+1} = f_i(x_i, \mu_i(x_i), w_i), \quad i = k+1, \ldots, N-1$$

where the first generated state is

$$x_{k+1} = f_k(x_k, u_k, w_k)$$

and each of the disturbances $w_k, \ldots, w_{N-1}$ is an independent random sample from the given distribution

- The costs corresponding to these trajectories are averaged to compute an approximation $\hat{Q}_k(X_k, u_k)$ to the Q-factor

$$E\left\{g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k))\right\}$$

416

- here, $\hat{Q}_k(X_k, u_k)$ is an approximation to $Q_k(X_k, u_k)$ because of the simulation error resulting from the use of a limited number of trajectories

- the approximation becomes increasingly accurate as the number of simulated trajectories increases

- once the approximate Q-factor $\hat{Q}_k(X_k, u_k)$ corresponding to each control $u \in U_k(x_k)$ is computed, we can obtain the (approximate) rollout control $\bar{\mu}_k(x_k)$ by the minimization

$$\bar{\mu}_k(x_k) = \arg \min_{u_k \in U_k(x_k)} \hat{Q}_k(x_k, u_k)$$

- there is, however, a serious issue with this approach, due to the simulation error involved in the calculation of the Q-factors

- in particular, for the calculation of $\bar{\mu}_k(x_k)$ to be accurate, the Q-factor differences

$$Q_k(x_k, u_k) - Q_k(x_k, \hat{u}_k)$$

must be computed accurately for all pairs of controls u_k and $\hat{u}_k$, so that these controls can be accurately compared

- let us now consider the last case of a stochastic problem and various possibilities for approximating the costs-to-go $H_k(x_k)$, $k = 1, dots, N^-1$, of the base policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$, rather than calculating them by Monte-Carlo simulation

- for example, in a certainty equivalence approach, given a state x_k at time k, we fix the remaining disturbances at some "typical" values $\bar{w}_{k+1}, \ldots, \bar{w}_{N-1}$, and we approximate the true Q-factor

$$Q_k(x_k, u_k) = E\left\{g_k(x_k, u_k, w_k) + H_{k+1}(f_k(x_k, u_k, w_k))\right\}$$

with

$$\tilde{Q}_k(x_k, u_k) = E\left\{g_k(x_k, u_k, w_k) + \tilde{H}_{k+1}(f_k(x_k, u_k, w_k))\right\}$$

where $\tilde{H}_{k+1} f_k(x_k, u_k, w_k))$ is obtained by

$$\tilde{H}_{k+1}(f_k(x_k, u_k, w_k)) = g_N(\bar{x}_N) + \sum_{i=k+1}^{N-1} g_i(\bar{x}_i, \mu_i(x_i), \bar{w}_i)$$

the initial state is

$$\bar{x}_{k+1} = f_k(x_k, u_k, w_k)$$

- the intermediate states are given by

$$\bar{x}_{i+1} = f_i(\bar{x}_i, \mu_i(x_i), \bar{w}_i), \quad i = k+1, \ldots, N-1$$

- thus, in this approach, the rollout control is approximated by

$$\bar{\mu}_k(x_k) = \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k)$$

- note that the approximate cost-to-go $\tilde{H}_{k+1}(x_{k+1})$ represents an approximation of the true cost-to-go $H_{k+1}(x_{k+1})$ of the base policy based on a single sample (the nominal disturbances $\bar{w}_{k+1}, \ldots, \bar{w}_{N-1}$)

- a potentially more accurate approximation is obtained using multiple nominal disturbance sequences and averaging the corresponding costs with appropriate nominal probabilities, similar to the scenario approximation approach of Example 5.3

Model Predictive Control

- in many control problems where the objective is to keep the state of a system near some desired point, the LQR models of showed previously are not satisfactory; there are two main reasons for this:

  a) The system may be nonlinear, and using for control purposes a model that is linearized around the desired point may be inappropriate.

  b) There may be control and/or state constraints, which are not handled adequately through a quadratic penalty on state and control. The reason may be special structure of the problem dictating that, for efficiency purposes, the system should often be operated at the boundary of its constraints. The solution obtained from a LQR model is not suitable for this, because the quadratic penalty on state and control tends to "blur" the boundaries of the constraints.

- these inadequacies of the linear-quadratic model have motivated a form of suboptimal control, called **model predictive control** (MPC), which combines elements of several ideas that we have discussed so far: certainty equivalent control, multistage lookahead, and rollout algorithms

- we will focus primarily on the most common form of MPC, where the system is either deterministic, or else it is stochastic, but it is replaced with a deterministic version by using typical values in place of all uncertain quantities, as in the certainty equivalent control approach

- at each stage, a (deterministic) optimal control problem is solved over a fixed length horizon, starting from the current state; the first component of the corresponding optimal policy is then used as the control of the current stage, while the remaining components are discarded; the process is then repeated at the next stage, once the next state is revealed

- as mentioned earlier, model predictive control (MPC) was initially motivated by the desire to introduce nonlinearities, and control and/or state constraints into the linear-quadratic framework, and obtain a suboptimal but stable closed-loop system

- with this in mind, we will describe MPC for the case of a stationary, possibly nonlinear, deterministic system, where state and control belong to some Euclidean spaces

- the system is
$$x_{k+1} = f(x_k, u_k), \quad k = 0, 1, \ldots,$$
and the cost per stage is quadratic:
$$x_k' Q x_k + u_k' R u_k, \quad k = 0, 1, \ldots,$$

  where Q and R are positive definite symmetric matrices

- we impose state and control constraints
$$x_k \in X, \quad u_k \in U(x_k), \quad k = 0, 1, \ldots,$$
and we assume that the set X contains the origin of the corresponding Euclidean space

- furthermore, if the system is at the origin, it can be kept there at no cost with control equal to 0, i.e., $0 \in U(0)$ and $f(0,0) = 0$

- we want to derive a stationary feedback controller that applies control $\bar{\mu}(x)$ at state x, and is such that, for all initial states $x_0 \in X$, the state of the closed-loop system

$$x_{k+1} = f(x_k, \bar{\mu}(x_k))$$

satisfies the state and control constraints, and the total cost over an infinite number of stages is finite

$$\sum_{k=0}^{\infty} (x_k' Q x_k + \bar{\mu}_k(x_k)' R \bar{\mu}_k(x_k)) < \infty$$

- note that because of the positive definiteness of Q and R, the feedback controller $\bar{\mu}$ is stable in the sense that $x_k \to 0$ and $\bar{\mu}(x_k) \to 0$ for all initial states $x_0 \in X$ (in the case of a linear system, the assumption of positive definiteness of Q may be relaxed to positive semidefiniteness, together with an observability assumption discussed previously)

- in order for such a controller to exist, it is evidently sufficient that there exists a positive integer m in such that for every initial state $x_0 \in X$, one can find a sequence of controls $u_k, k = 0, 1, \ldots, m-1$ which drive to 0 the state x_m of the system at time m, while keeping all the preceding states $x_1, x_2, \ldots, x_{m-1}$ within X and satisfying the control constraints $u_0 \in U(x_0), \ldots, u_{m-1} \in U(x_{m-1})$

- we refer to this as the constrained controllability assumption; in practical applications, this assumption can often be checked easily

- let us now describe a form of MPC under the preceding assumption: at each stage k and state $x_k \in X$, it solves an m-stage deterministic optimal control problem involving the same quadratic cost and the requirement that the state after m stages be exactly equal to 0

- this is the problem of minimizing
$$\sum_{i=k}^{k+m-1} (x_i' Q x_i + u_i' R u_i)$$
subject to the system equation constraints
$$x_{i+1} = f(x_i, u_i), \quad i = k, k+1, \ldots, k+m-1,$$
the state and control constraints
$$x_i \in X, \quad u_i \in U(x_i), \quad i = k, k+1, \ldots, k+m-1,$$
and the terminal state constraint $x_{k+m} = 0$

- by the constrained controllability assumption, this problem has a feasible solution

- let $\{\bar{u}_k, \ldots, \bar{u}_{k+m-1}\}$ be a corresponding optimal control sequence; the MPC applies at stage k the first component of this sequence
$$\bar{\mu}(x_k) = \bar{u}_k$$
and discards the remaining components

425

- regarding the choice of the horizon length m for the MPC calculations, note that if the constrained controllability assumption is satisfied for some value of m, it is also satisfied for all larger values of m (and can be shown to provide better results)

- on the other hand, the optimal control problem solved at each stage by the MPC becomes larger and hence more difficult as m increases; thus, the horizon length is usually chosen on the basis of some experimentation

- the MPC scheme that we have described is just the starting point for a broad methodology with many variations, which often relate to the sub-optimal control methods that we have discussed so far in this chapter

Discretization Issues

- an important practical issue is how to deal computationally with problems involving nondiscrete state and control spaces - in particular, problems with continuous state, control or disturbance spaces must be discretized in order to execute the DP algorithm

- here each of the continuous spaces of the problem is replaced by a space with a finite number of elements, and the system equations are appropriately modified

- thus the resulting approximating problem involves a finite number of states and a set of transition probabilities between these states

- once the discretization is done, the DP algorithm is executed to yield the optimal cost-to-go function and an optimal policy for the discrete approximating problem

- the optimal cost function and/or the optimal policy for the discrete problem may then be extended to an approximate cost function or a suboptimal policy for the original continuous problem through some form of interpolation (we have seen examples of such procedures)

- a prerequisite for success of this type of discretization is **consistency**

- by this we mean that the optimal cost of the original problem should be achieved in the limit as the discretization becomes finer and finer

- consistency is typically guaranteed if there is a "sufficient amount of continuity" in the problem; for example, if the cost-to-go functions of and the optimal policy of the original problem are continuous functions of the state

- this in turn can be guaranteed through appropriate assumptions on the original problem data

- if the original problem is defined in continuous time, then the time must be also discretized, to obtain a discrete-time approximating problem

- the issue of consistency becomes now considerably more complex, because the time discretization affects not only the system equations but also the control constraint set

- in particular, the control constraint set may change considerably as we pass to the appropriate discrete-time approximation

- as an example, consider the two-dimensional system

$$\dot{x}_1(t) = u_1(t), \quad \dot{x}_2(t) = u_2(t),$$
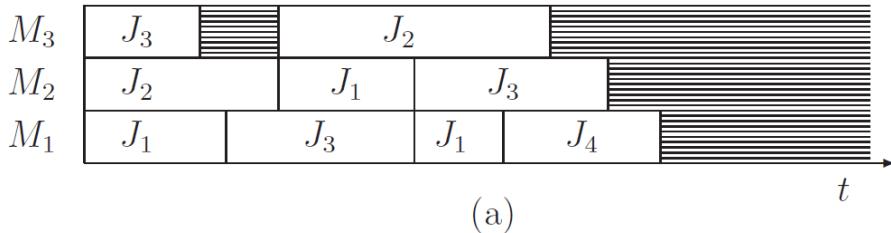
with the control constraint

$$u_1(t) \in \{-1, 1\}, \quad u_2(t) \in \{-1, 1\}$$
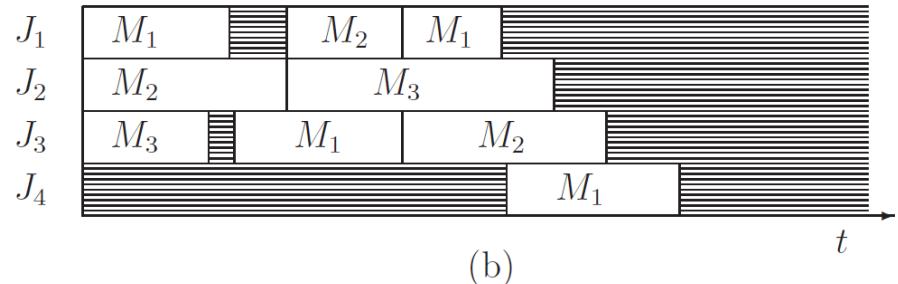
- it can be seen then that the state at time $t + \Delta t$ can be anywhere within the square centered at $x(t)$ with side length $2\Delta t$ (note that the effect of any control in the interval $[-1, 1]$ can be obtained in the continuous system by "chattering" between $+1$ and -1 controls)

- thus, given Δt, the appropriate discrete-time approximation of the control constraint set should involve a discretized version of the entire unit square, the convex hull of the control constraint set of the continuous-time problem

- as another example, think how we would discretize the problem in Example 5.5

8   Scheduling

- the theory of scheduling is characterized by a virtually unlimited number of problem types and applications

- in the following slides, we will cover a basic classification for the scheduling problems, which is (in slight variations) widely used in the literature

- suppose that m machines $M_j(j = 1, \ldots, m)$ have to process n jobs $J_i(i = 1, \ldots, n)$

- a schedule is for each job an allocation of one or more time intervals to one or more machines

- schedules may be represented by Gantt charts as shown in the following figure (Gantt charts may be machine-oriented or job-oriented)

- the corresponding scheduling problem is to find a schedule satisfying certain restrictions

(a)   (b)

Job data:

- a job J_i consists of a number n_i of operations $O_{i1}, \ldots, O_{i,n_i}$

- associated with operation O_{ij} is a processing requirement p_{ij}

- a release date r_i, on which the first operation of J_i becomes available for processing may be specified

- associated with each operation O_{ij} is a set of machines $\mu_{ij} \subseteq \{M_1, \ldots, M_m\}$ on which the operation can be performed (μ_{ij} are one element sets – dedicated machines; μ_{ij} are the set of all machines – parallel machines; neither of the two cases – multi-purpose machines)

- it is also possible that all machines in the set μ_{ij} are used simultaneously by O_{ij} during the whole processing period – multiprocessor task scheduling problems

- a cost function $f_i(t)$ measures the cost of completing J_i at time t, where a due date d_i and a weight w_i may be used in defining f_i

- in general, all data $p_i, p_{ij}, r_i, d_i, w_i$ are assumed to be integer

- a schedule is feasible if no two time intervals overlap on the same machine, if no two time intervals allocated to the same job overlap, and if, in addition, it meets a number of problem-specific characteristics

- a schedule is optimal if it minimizes a given optimality criterion

- there is a large variety of classes of scheduling problems which differ in their complexity and in the algorithms that are used for solving them

- classes of scheduling problems are specified in terms of a three-field classification $\alpha|\beta|\gamma$ where α specifies the machine environment, β specifies the job characteristics, and γ denotes the optimality criterion

Job characteristics:

- specified by a set β containing at the most six elements $\beta_1, \ldots, \beta_6$

- β_1 indicates whether preemption (or job splitting) is allowed – if it is allowed, we set $\beta_1 = pmtn$; otherwise β_1 does not appear in β

- β_2 describes precedence relations between jobs, which are usually represented by an acyclic directed graph $G = (V, A)$ – if G is an arbitrary acyclic directed graph we set $\beta_2 = prec$, otherwise the graph type is described in detail (e.g. chains, trees, series-parallel directed graphs, etc)

- if $\beta_3 = r_i$, then release dates may be specified for each job, otherwise if $r_i = 0$ for all jobs, then β_3 does not appear in β

- β_4 specifies restrictions on the processing times or on the number of operations – if β_4 is equal to $p_i = 1$ ($p_{ij} = 1$), then each job (operation) has a unit processing requirement

- if $\beta_5 = d_i$, then a deadline d_i is specified for each job J_i, i.e. job J_i must finish not later than time d_i

- in some applications, sets of jobs must be grouped into batches – β_6 describes whether the problem includes batching

Machine environment:

- characterized by a string $\alpha = \alpha_1 \alpha_2$ of two parameters, where possible values for α_1 are $\circ, P, Q, R, PMPM, QMPM, G, X, O, J, F$

- if $\alpha_1 \in \{\circ, P, Q, R, PMPM, QMPM\}$, then each J_i has a single operation

- if $\alpha_1 = \circ$ (empty symbol), each job must be processed on a specified dedicated machine

- if $\alpha_1 \in \{P, Q, R\}$, then we have parallel machines (each job can be processed on each of the machines), where P denotes identical machines ($p_{ij} = p_i$), Q uniform machines ($p_{ij} = p_i/s_j$), R unrelated machines ($p_{ij} = p_i/s_{ij}$)

- if $\alpha_1 = PMPM$ and $\alpha_1 = QMPM$, then we have multi-purpose machines with identical and uniform speeds, respectively

- if $\alpha_1 \in \{G, X, O, J, F\}$, we have a multi-operation model (associated with each J_i there is a set of operations $O_{i1}, \ldots, O_{i,n_i}$), the machines are dedicated (μ_{ij} are one element sets) – general, job shop, flow shop, etc.

- α_2 encodes the number of machines

435

Optimality criteria:

- denote the finishing time of job J_i by C_i and the associated cost by $f_i(C_i)$
  - there are essentially two types of total cost functions

  $$f_{\max}(C) = \max\{f_i(C_i) \mid i = 1, \ldots, n\} \qquad \text{and} \qquad \sum f_i(C) = \sum_{i=1}^{n} f_i(C_i)$$
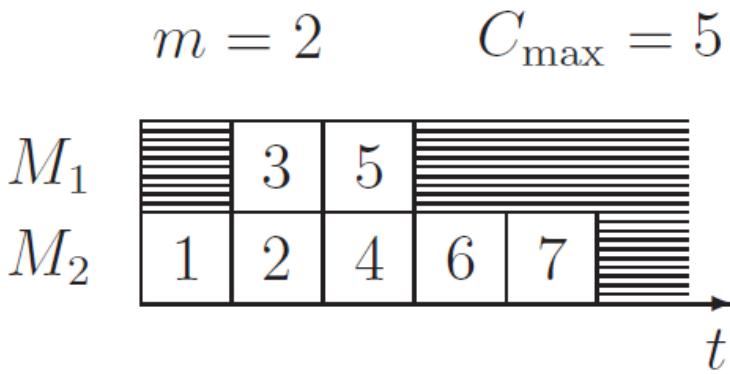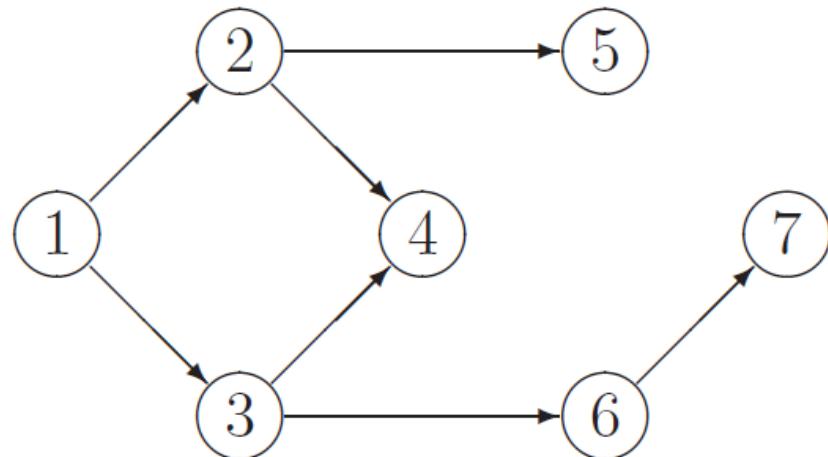
  called bottleneck objectives and sum objectives, respectively

- if the functions f_i are not specified, we set $\gamma = f_{\max}$ or $\gamma = \sum f_i$

- the most common objective functions are makespan $\max\{C_i \mid i = 1, \ldots, n\}$, total flow time $\sum_{i=1}^{n} C_i$, and weighted (total) flow time $\sum_{i=1}^{n} w_i C_i$; in these cases we write $\gamma = C_{\max}, \gamma = \sum C_i, \gamma = \sum w_i C_i$

- other objective functions may depend on d_i: lateness $(L_i = C_i - d_i)$, earliness $(E_i = \max\{0, d_i - C_i\})$, tardiness $(T_i = \max\{0, C_i - d_i\})$,

  unit penalty $U_i = \begin{cases} 0 & \text{if } C_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$, etc.

- linear combinations of the different objective functions are also used

Example 8.1 – Various Problems
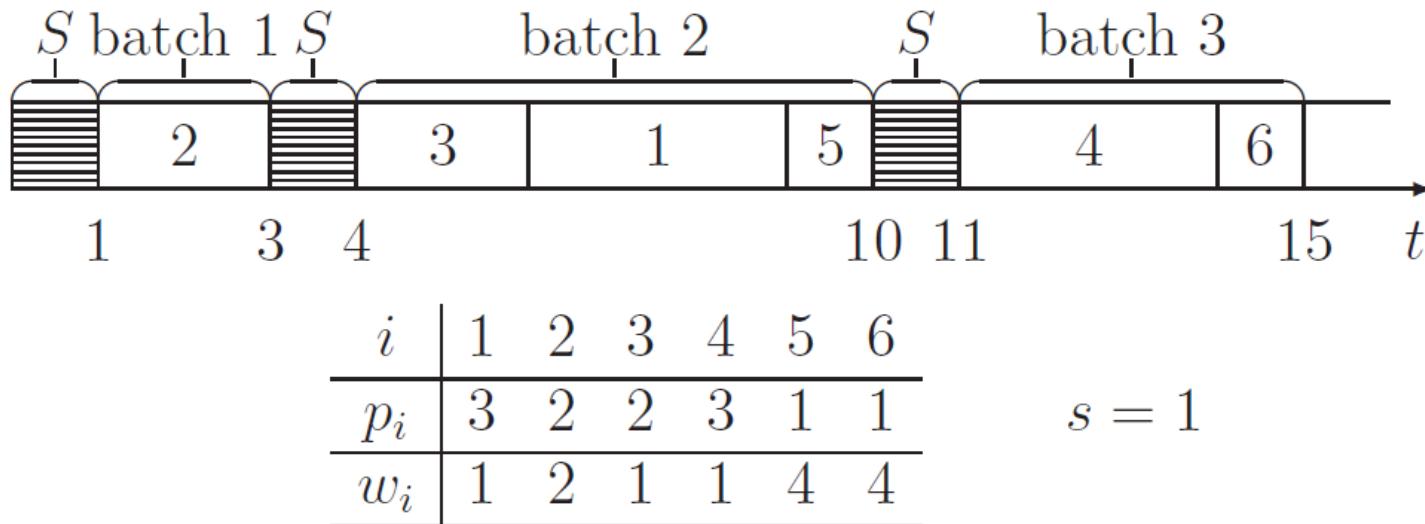
$P||prec; p_i = 1||C_{\max}$

- is the problem of scheduling jobs with unit processing times and arbitrary precedence constraints on m identical machines such that the makespan is minimized

- an instance is given by a directed graph with n vertices and the number of machines

$$m = 2 \qquad C_{\max} = 5$$

$1||s\text{-}batch|| \sum w_i C_i$

- is the problem of splitting a set of jobs into batches and scheduling these batches on one machine such that the weighted flow time is minimized

- the processing time of a batch is the sum of processing times of the jobs in the batch

| i	1	2	3	4	5	6	
p_i	3	2	2	3	1	1	$s = 1$
w_i	1	2	1	1	4	4	

$$\sum w_i C_i = 2 \cdot 3 + (1 + 1 + 4) \cdot 10 + (1 + 4)15$$

438
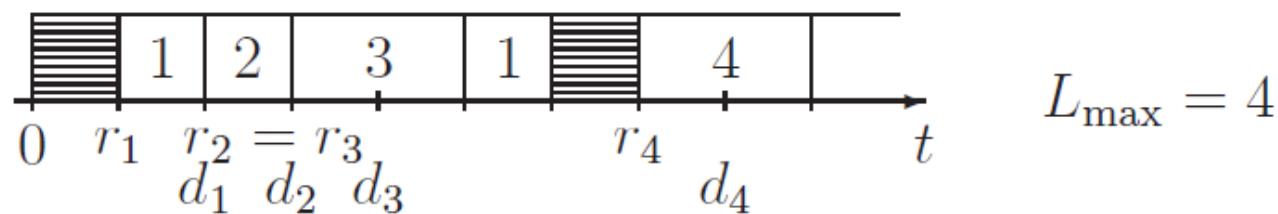
$1||r_i; pmtn||L_{\max}$

- is the problem of finding a preemptive schedule on one machine for a set of jobs with given release times $r_i \neq 0$ such that the maximum lateness is minimized
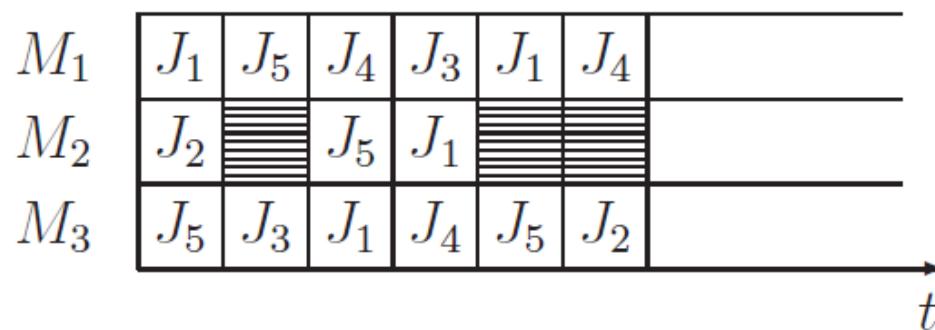
| i	1	2	3	4
p_i	2	1	2	2
r_i	1	2	2	7
d_i	2	3	4	8

$L_{\max} = 4$

$J3||p_{ij} = 1||C_{\max}$

- is the problem of minimizing maximum completion time in a three-machine job shop with unit processing times

i/j	1	2	3	4
1	M_1	M_3	M_2	M_1
2	M_2	M_3	—	—
3	M_3	M_1	—	—
4	M_1	M_3	M_1	—
5	M_3	M_1	M_2	M_3

Solution Techniques

- some scheduling problems can be solved efficiently by reducing them to well-known combinatorial optimization problems and using the corresponding solution techniques:

    - linear programs

    - maximum flow problems $(O||pmtn||C_{\max})$

    - assignment problems $(1||r_i; p_i = 1|| \sum f_i)$

    - transportation problems

- others can be solved by using standard techniques:

    - dynamic programming $(P|| \sum w_i C_i)$

    - branch-and-bound methods $(F2|| \sum C_i$ – Example 2.10$)$

Complexity Classes

- a computational problem can be viewed as a function h that maps each input x in some given domain to an output $h(x)$ in some given range

- we are interested in algorithms for solving computational problems – such an algorithm computes $h(x)$ for each input x

- one of the main issues of complexity theory is to measure the performance of algorithms with respect to computational time – for each input x we define the input length $|x|$ as the length of some encoding of x

- we measure the efficiency of an algorithm by an upper bound $T(n)$ on the number of steps that the algorithm takes on any input x with $|x| = n$

- in most cases it is hard to calculate the precise form of T, and we will replace the precise form of T by its asymptotic order – we say that $T(n) \in O(g(n))$ if there exist constants $c > 0$ and a nonnegative integer n_0 such that $T(n) \leq cg(n)$ for all integers $n \geq n_0$

- this means that rather than saying that the computational complexity is bounded by $7n^3 + 27n^2 + 4$, we say simply that it is $O(n^3)$

- as an example, consider the problem $1|| \sum w_i C_i$:
  - the input x for this problem is given by the number n of jobs and two n-dimensional vectors (p_i) and (w_i)
  - define $|x|$ to be the length of a binary encoded input string for x
  - the output $f(x)$ for the problem is a schedule minimizing $\sum_{i=1}^{n} w_i C_i$ (it can be represented by an n-vector of all C_i-values)
  - the following algorithm calculates these C_i-values (based on the interchange argument):
    1. enumerate the jobs such that: $w_1/p_1 \geq w_2/p_2 \geq \cdots \geq w_n/p_n$
    2. $C_0 = 0$
    3. for $i = 1$ to n do : $\quad C_i = C_{i-1} + p_i$
  - the number of computational steps in this algorithm can be bounded as follows: in Step 1 the jobs have to be sorted – this takes $O(n \log n)$ steps; Step 3 can be done in $O(n)$ time.
  - thus, we have $T(n) \in O(n \log n)$ (if we replace n by the input length $|x|$, the bound is still valid because we always have $n \leq |x|$)

443

- a problem is called polynomially solvable if there exists a polynomial p such that $T(|x|) \in O(p(|x|))$ for all inputs x for the problem, i.e. if there is a k such that $T(|x|) \in O(|x|^k)$

- the notion polynomially solvable depends on the encoding – we assume that all numerical data describing the problem are binary encoded

- for example, the dynamic programming algorithm for the knapsack problem, which runs in $O(nW)$ time, is not polynomially bounded because the number of steps depends on W, which is an exponentially growing function of the length of an input string with a binary encoding

- this algorithm is called pseudopolynomial which means that $T(n)$ is polynomial where n is the input length with respect to unary encoding (with unary encoding an integer d is represented by a sequence of d ones)

- a problem is called pseudopolynomially solvable if there exists a pseudopolynomial algorithm which solves the problem

- a problem is called a decision problem if the output range is $\{yes, no\}$

- we may associate with each scheduling problem a decision problem by defining a threshold k for the corresponding objective function f

- this decision problem is: Does there exist a feasible schedule S such that $f(S) \leq k$?

- the class of all decision problems which are polynomially solvable is denoted by $\mathcal{P}$

- when a scheduling problem is formulated as a decision problem there is an important asymmetry between those inputs whose output is "yes" and those whose output is "no" – a "yes"-answer can be certified by a small amount of information: the feasible schedule S with $f(S) \leq k$

- given this certificate, the "yes"-answer can be verified in polynomial time; this is not the case for the "no"-answer

- in general, let $\mathcal{NP}$ denote the class of decision problems where each "yes" input x has a certificate y, such that $|y|$ is bounded by a polynomial in $|x|$ and there is a polynomial-time algorithm to verify that y is a valid certificate for x

- it can be shown that other scheduling problems when considered as decision problems belong to $\mathcal{NP}$

- every decision problem solvable in polynomial time belongs to $\mathcal{NP}$

- if we have such a problem P and an algorithm which calculates for each input x the answer $h(x) \in \{yes, no\}$ in a polynomial number of steps, then this answer $h(x)$ may be used as a certificate (this certificate can be verified by the algorithm), thus P is also in $\mathcal{NP}$ which implies $\mathcal{P} \subseteq \mathcal{NP}$

- one of the major open problems of modern mathematics is whether $\mathcal{P}$ equals $\mathcal{NP}$ – it is generally conjectured that this is not the case

- the problems in $\mathcal{NP}$ can be grouped into so-called $\mathcal{NP}$-hard and $\mathcal{NP}$-complete problems

- the knowledge that a scheduling problem is $\mathcal{NP}$-hard is little consolation for the algorithm designer who needs to solve the problem

- fortunately, despite theoretical equivalence, not all $\mathcal{NP}$-hard problems are equally hard from a practical perspective: as we have seen in the knapsack example (that is $\mathcal{NP}$-hard), dynamic programming and branch-and-bound methods can be sometimes applied to solve reasonably-sized problems

- another possibility is to apply approximation algorithms, local search methods, simulated annealing, tabu search, various other heuristics or metaheuristics, in the cases where the more traditional approaches fail, and try to get at least a very good suboptimal solutions

Complexity of single machine problems:

$1 \mid prec; r_j \mid C_{\max}$	$O(n^2)$
$1 \mid prec; r_j; p_j = p \mid L_{\max}$	$O(n^3 \log \log n)$
$1 \mid prec \mid f_{\max}$	$O(n^2)$
$1 \mid prec; r_j; p_j = 1 \mid f_{\max}$	$O(n^2)$
$1 \mid prec; r_j; pmtn \mid f_{\max}$	$O(n^2)$
$1 \mid r_j; pmtn \mid \sum C_j$	$O(n \log n)$
$1 \mid prec; r_j; p_j = p \mid \sum C_j$	$O(n^2)$
$1 \mid prec; r_j; p_j = p, pmtn \mid \sum C_j$	$O(n^2)$
$1 \mid r_j; p_j = p \mid \sum w_j C_j$	$O(n^7)$
$1 \mid sp\text{-graph} \mid \sum w_j C_j$	$O(n \log n)$
$1 \mid\mid \sum U_j$	$O(n \log n)$
$1 \mid r_j; pmtn \mid \sum U_j$	$O(n^5)$
	$O(n^4)$
$1 \mid r_j; p_j = p \mid \sum w_j U_j$	$O(n^7)$
$1 \mid r_j; p_j = p; pmtn \mid \sum w_j U_j$	$O(n^{10})$
$1 \mid r_j; p_j = p \mid \sum T_j$	$O(n^7)$
$1 \mid r_j; p_j = 1 \mid \sum f_j$	$O(n^3)$
$1 \mid r_j; p; pmtn \mid \sum T_j$	$O(n^2)$

Polynomially solvable single machine problems.

$1 \mid\mid \sum w_j U_j$

$1 \mid r_j; pmtn \mid \sum w_j U_j$

$1 \mid\mid \sum T_j$

Pseudopolynomially solvable single machine problems.

$1 \mid r_j \mid L_{\max}$

$1 \mid r_j \mid \sum C_j$

$1 \mid prec \mid \sum C_j$

$1 \mid chains; r_j; pmtn \mid \sum C_j$

$1 \mid prec; p_j = 1 \mid \sum w_j C_j$

$1 \mid chains; r_j; p_j = 1 \mid \sum w_j C_j$

$1 \mid r_j; pmtn \mid \sum w_j C_j$

$1 \mid chains; p_j = 1 \mid \sum U_j$

$1 \mid\mid \sum w_j U_j$

$1 \mid\mid \sum T_j$

$1 \mid chains; p_j = 1 \mid \sum T_j$

$1 \mid\mid \sum w_j T_j$

$\mathcal{NP}$-hard single machine problems.

(more info on the different problems/algorithms/etc. can be found in the Bruckner book)

Example 8.2 – $1 || \sum w_i U_i$

- given n jobs $i = 1, \ldots, n$ with processing times p_i and due dates d_i, schedule these jobs such that $\sum_{i=1}^{n} w_i U_i$ is minimized where $w_i \geq 0$

- assume that the jobs are enumerated according to nondecreasing d_i:

$$d_1 \leq d_2 \leq \cdots \leq d_n$$

- then there exists an optimal schedule given by a sequence of the form

$$i_1, i_2, \ldots, i_s, i_{s+1}, \ldots, i_n$$

  where jobs $i_1 < i_2 < \cdots < i_s$ are on time and jobs $i_s, \ldots, i_n$ are late

- interchange argument: If a job i is late, we may put i at the end of the schedule without increasing the objective function. If i and j are early jobs with $d_i \leq d_j$ such that i is not scheduled before j, then we may shift the block of all jobs scheduled between j and i jointly with i to the left by p_j time units and schedule j right after this block. Because i was not late and $d_i \leq d_j$ this creates no late job.

449

- to solve the problem, we calculate for $t = 0, 1, \ldots, T = \sum_{i=1}^{n} p_i$ and $j = 1, \ldots, n$ the minimum criterion value $F_j(t)$ for the first j jobs subject to the constraint that total processing time of the on-time jobs is at the most t

- if $0 \leq t \leq d_j$ and job j is on time in a schedule witch corresponds with $F_j(t)$, then $F_{j-1}(t - p_j)$; otherwise $F_j(t) = F_{j-1}(t) + w_j$

- if $t > d_j$, then $F_j(t) = F_j(d_j)$ because all jobs $1, 2, \ldots, j$ finishing later than $d_j \geq \cdots \geq d_1$ are late

- thus, for $j = 1, \ldots, n$ we have a forward DP recursion

$$F_j(t) = \begin{cases} \min\{F_{j-1}(t - p_j), F_{j-1}(t) + w_j\} & \text{for } 0 \leq t \leq d_j \\ F_j(d_j) & \text{for } d_j < t \leq T \end{cases}$$

  with $F_j(t) = \infty$ for $t < 0, j = 0, \ldots, n$ and $F_0(t) = 0$ for $t \geq 0$.

- notice that $F_n(d_n)$ is the optimal solution to the problem

9   What next?

- we did not cover a lot of the more advanced topics:

  - how to do approximations efficiently – if the state/control/uncertainty spaces are too big, or if we need to discretize continuous spaces (roll-out policies and stochastic programming, functional/parametric approximations, Q-learning, reinforcement learning)

  - how to deal with additional constraints on control in LQRs and with different kinds of uncertainty (robust/stochastic model predictive control)

- solid theoretical/application work in the above-mentioned areas is worth a PhD

Additional resources:

- Berkeley course: Advanced robotics

  `https://people.eecs.berkeley.edu/~pabbeel/cs287-fa19/`

- Warren B. Powell: Reinforcement Learning and Stochastic Optimization, 2019; or Approximate Dynamic Programming, 2011

  `https://castlelab.princeton.edu/RLSO/`

- Dimitris Bertsekas: Rollout, Policy Iteration, and Distributed Reinforcement Learning, 2020; or Reinforcement Learning and Optimal Control, 2019

- Basil Kouvaritakis and Mark Cannon: Model Predictive Control (Classical, Robust and Stochastic), 2016