Chapter 2

Deterministic Problems

[1] In this chapter, we focus on deterministic control problems (with perfect information), i.e, there is no disturbance w_k in the dynamics equation. For such problems there is no advantage from the point of view of cost optimization in using a closed-loop feedback control policy instead of an open-loop policy computed at the initial time. We review this fact in the first section. In addition, for deterministic problem we can execute the DP algorithm *forward* in time, a fact which can be useful for real-time applications. The forward and backward DP algorithms have also various applications in algorithm design, in particular to compute shortest paths in graphs. The rest of the chapter describes a few of these applications.

2.1 The Value of Information: Open-Loop vs. Closed-Loop Control

Recall from chapter 1 that our objective, now *in the absence of disturbances or measurement uncertainty*, is to minimize

$$J_0(x_0, \pi) = c_N(x_N) + \sum_{k=0}^{N-1} c_k(x_k, \mu_k(x_k)), \tag{2.1}$$

over all policies $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$, subject to the dynamics

$$x_0 \text{ given, } x_{k+1} = f_k(x_k, u_k).$$

Recall that the control a time k, $u_k = \mu_k(x_k)$, is allowed to depend on the state at time k [2], i.e., this choice of control can depend on information obtained while the system is running. These policies are called *closed-loop* or *feedback*

[1] This version: September 12 2009

[2] recall that by definition of the state, there is no point in storing more information than x_k about the past trajectory in order to choose the control u_k. See also [Ber07, vol. II, p. 10]

policies. Generating a feedback policy requires sensors to observe the trajectory of the system and on-line computations of the controls based on the sensor measurements. An alternative, usually less demanding setup, and one that cannot be avoided if the system trajectory cannot be observed, is to ask that a given set of fixed control inputs $u_0, u_1, \ldots, u_{N-1}$ (*vectors*, not *functions* as in our feedback policies) be computed once and for all at time $t = 0$. The resulting policy is such that $\mu_k(x_k) = u_k$ is independent of the value of the state x_k, and is called an *open-loop* policy. An optimal open-loop policy can be computed for deterministic problems by solving the *optimization* problem

$$J_0(x_0, \pi) = \min_{u_0,\ldots,u_{N-1}} \left[c_N(x_N) + \sum_{k=0}^{N-1} c_k(f^{(k)}(x_0, u_0, \ldots, u_{k-1}), u_k) \right], \quad (2.2)$$

where $f^k(x_0, u_0, \ldots, u_{k-1})$ is defined recursively as

$$f^{(0)}(x_0) = x_0, \; f^{(k)}(x_0, u_0, \ldots, u_{k-1}) = f_{k-1}(f^{(k-1)}(x_0, u_0, \ldots, u_{k-2}), u_{k-1}).$$

That is, x_0 and a given choice of controls $u_{0:N-1}$ determines the trajectory $x_{1:N}$ of the system and we can just propagate the dynamics in the cost function to express it only in terms of x_0 and our control inputs. For systems subject to perturbations, one can obtain significant improvements by using closed-loop policies over open-loop policies. In fact, in the control of physical systems, completely open-loop policies rarely make sense because disturbances and model uncertainties are unavoidable and can have disastrous consequences if not properly taken into account (see problem set 1). Nonetheless, because open-loop policies are sometimes easier to compute than closed-loop policies, or can be computed offline before the system starts, which is advantageous for real-time applications, they can form the basis of heuristics which periodically adjust the policy while running open loop most of the time.

In the ideal set-up where disturbances are absent, closed-loop policies do not in fact achieve a smaller cost than open-loop policies. This comes from our remark that a set of control inputs completely determines the future trajectory of the system, not just probabilistically but deterministically. Once u_0 is chosen, this determines x_1 and there is no advantage in waiting until the next time step to decide the control input u_1, etc. Hence for deterministic problems, there is a variety of techniques available to minimize (2.1). One can use the DP algorithm or try to solve directly the (large) optimization problem (2.2). The DP algorithm will provide a closed-loop policy: i.e., a sequence of *functions* $\mu_k(x_k)$. In contrast, the optimization method will produce a sequence of controls $u = [u_0, \ldots, u_{N-1}]$. This optimization problem needs to be solved again if one wants to obtain a new sequence for a different value of the initial condition x_0, although specialized optimization techniques such as "warm start" can help alleviate this problem (see chapter 9). From a closed-loop policy, one can easily extract an open-loop policy once the initial condition x_0 is fixed by propagating the dynamics, using $u_0 = \mu_0(x_0)$, $u_1 = \mu_1(f_0(x_0, \mu_0)), \ldots$ Since closed-loop and open-loop policies have the same performance in this case, from a mathematical point of view, if x_0 is fixed, a closed-loop policy contains a lot more information that is necessary. However in practice, it can still be used on a system that deviated from its deterministic trajectory due to unmodelled disturbances, while still maintaining the optimality for the remaining periods after deviation. Using an unmodified sequence of open-loop controls on a system whose trajectory has been perturbed can be a very bad idea (again, see problem set 1), and maybe in fact infeasible since the control constraint sets $\mathsf{U}_k(x_k)$ depend on the state.

Let us mention an alternative optimization technique, in fact typically preferable to (2.2), to solve the deterministic optimal-control problem. Instead of propagating the dynamics as in (2.2), we consider the problem as an *equality constrained optimization problem*

$$\begin{aligned} J^*(x_0) = \min c_N(x_N) + \sum_{k=0}^{N-1} c_k(x_k, u_k) \qquad\qquad (2.3)\\ \text{subject to } x_{k+1} = f_k(x_k, u_k), k = 0, \dots, N-1 \\ u_k \in \mathsf{U}_k(x_k). \end{aligned}$$

In this optimization problem we can take as our decision variables all the variables $u_0, \dots, u_{N-1}, x_1, \dots, x_N$. If the control and state variables are continuous, and the set $U_k(x_k)$ has a description say in terms of inequality constraints, then once can use an off-the-shelf nonlinear programming solver.

Exercise 4. When is the optimization problem (2.3) convex?

The constrained optimization point of view also leads to a discrete-time version of the *minimum principle* (a term more often used for continuous-time systems, which we do not cover in this course), see [Ber07, Vol. I, p.129]. Let us assume that $\mathsf{X}_k = \mathbb{R}^n$, $\mathsf{U}_k(x_k) = \mathbb{R}^m$ for all k (in particular, the controls are unconstrained). Since we will use differential calculus, we assume sufficient differentiability properties when necessary. Let us adjoin the dynamics constraints to the objective using the Lagrange multipliers $\lambda_1, \dots \lambda_N$ to form the Lagrangian:

$$\begin{aligned} L(\mathbf{x}, \mathbf{u}, \lambda) &= c_N(x_N) + \sum_{k=0}^{N-1} c_k(x_k, u_k) + \sum_{k=0}^{N-1} \lambda_{k+1}^T (f_k(x_k, u_k) - x_{k+1}) \\ &= c_N(x_N) + \sum_{k=0}^{N-1} c_k(x_k, u_k) + \lambda_{k+1}^T (f_k(x_k, u_k) - x_{k+1}), \end{aligned}$$

with $\mathbf{x} = (x_1, \dots, x_N)$, $\mathbf{u} = (u_0, \dots, u_{N-1})$. Then if $(\mathbf{x}, \mathbf{u})$ is optimal, the gradient of the Lagrangian with respect to $(\mathbf{x}, \mathbf{u})$ and with respect to λ must be 0 (necessary condition). Define the "Hamiltonian" function

$$H_k(x_k, u_k, \lambda_{k+1}) = c_k(x_k, u_k) + \lambda_{k+1}^T f_k(x_k, u_k), k = 0, \dots, N-1.$$

The condition $\nabla_{u_k} L(\mathbf{x}, \mathbf{u}, \lambda) = 0$ gives

$$\nabla_{u_k} H_k = \nabla_{u_k} c_k(x_k, u_k) + \left(\frac{\partial f_k}{\partial u_k} \right)^T \lambda_{k+1} = 0, \tag{2.4}$$

where $\partial f_k / \partial u_k$ is the (partial) Jacobian matrix of f_k as a function of u_k. The condition $\nabla_{x_k} L(\mathbf{x}, \mathbf{u}, \lambda) = 0$ gives the backward recursion (note that the sum in the Lagrangian has two terms involving x_k)

$$\lambda_k = \nabla_{x_k} H_k = \nabla_{x_k} c_k(x_k, u_k) + \left(\frac{\partial f_k}{\partial x_k} \right)^T \lambda_{k+1}, \tag{2.5}$$

called the *discrete-time adjoint equation*, with terminal condition

$$\lambda_N = \nabla_{x_N} c_N(x_N). \tag{2.6}$$

The variable λ is sometimes called the *co-state*. For a problem where u_k is constrained to belong to a set $\mathsf{U}_k(x_k)$ and this set is *convex*, some knowledge of convex analysis or a picture will tell you that (2.4 must be replaced by

$$\nabla_{u_k} H_k(x_k^*, u_k^*, \lambda_{k+1})(u_k - u_k^*) \geq 0, \forall u_k \in \mathsf{U}_k(x_k^*), \tag{2.7}$$

where u_k^* and x_k^* are an optimal input and a state on the corresponding optimal trajectory. If in addition H_k is a convex function of u_k for any fixed x_k and λ_{k+1}, then (2.7) is also a sufficient condition to characterize the minimum of H_k with respect to u_k so in that case

$$u_k^* \in \arg \min_{u_k \in \mathsf{U}_k(x_k^*)} H_k(x_k^*, u_k, \lambda_{k+1}).$$

It is good to remember that the minimum principle (2.7), (2.5), (2.6) is a *necessary* condition for optimality, similar to a first-order optimality condition in classical optimization. In successful applications, the minimum principle is used to isolate a small set of possible candidate optimal solutions. For example, if we have a unique candidate and we know by other means that the problem must have a minimum, then this candidate is the optimum. In contrast, dynamic programming leads to a *sufficient* solution for optimality. Finally, it is interesting to note that for continuous-time problems, the convexity assumption on U_k is not needed. The intuitive reason is that we can always "convexify" the set of allowed velocity directions $f_k(x_k, u_k)$ at x_k by using controls switching arbitrarily fast (chattering controller).

2.2 Deterministic DP and Algorithm Design

You might be already familiar with the dynamic programming principle from a previous algorithm course. In this section, we explore the use of the DP algorithm to solve various problems more efficiently, and in particular its connection to shortest path problems.

Dynamic Programming in Combinatorial Optimization

Dynamic programming is a fundamental tool to solve certain combinatorial optimization problems in polynomial time, and in other cases it can be used to speed-up computations compared to naive enumeration, even if it does not achieve polynomial-time performance, see [BT97, section 11.3]. One such problem is the famous *traveling salesman problem* (TSP), which is NP-complete. In the TSP we have n cities and we are given the distances d_{ij} between each pair (i,j) of cities. We want to find a minimum length tour that visits every city once and returns to the origin city. Let us call the origin city 1. To solve the TSP using dynamic programming, we define as the state the set S of cities already visited as well as the current location $l \in S$. The state (S,l) can be reached from any state of the form $(S \setminus \{l\}, m)$ with $m \in S \setminus \{l\}$, for a traveling cost c_{ml}. We immediately get the recursion for the optimal cost of visiting S and finishing at $l \in S$:

$$C(S,l) = \min_{m \in S \setminus \{l\}} \Big\{ C(S \setminus \{l\}, m) + c_{ml} \Big\}, \forall l \in S,$$

and the initialization $C(\{1\},1) = 0$. Finally, the length of the optimum tour is given by $\min_l \Big\{ C(\{1,\ldots,n\}, l) + c_{l1} \Big\}$. This algorithm of course still runs in exponential time, in fact in time $O(n^2 2^n)$. Note that the reason why the DP algorithm does not lead to tractable algorithms for hard problems is in general due to the number of states growing exponentially, a problem that we will encounter frequently in various settings. Still, DP is much better than naive enumeration of all $n!$ tours of cities (recall Stirling's formula: $n! \sim (n/e)^n \sqrt{2\pi n}$).

Exercise 5. Give the details on how to solve TSP in time $O(n^2 2^n)$.

Deterministic Finite-State Finite-Horizon Optimal Control Problems as Shortest Path Problems

Consider a deterministic problem as described in section 2.1. Let us now assume that the state space X_k is finite, for all k. Then the control problem (2.1) can be solved by *solving a shortest path problem in a graph* that is constructed as follows (see Fig. 2.1). The graph can be decomposed into levels, with one level per time period $k = 0, 1, \ldots, N$. There is an additional artificial terminal node t to which point all the nodes of the last level $k = N$. A node in the graph at level k corresponds to a value of the state at time k. Hence there are $|\mathsf{X}_k|$ nodes at level k ($|S|$ denotes the cardinal of a set S). Nodes at level k have edges to nodes at levels $k+1$. A node corresponding to state value x_k has an edge to a node with state value x_{k+1} if and only if $x_{k+1} = f_k(x_k, u_k)$, i.e., there is a control value u_k which drives the system from system from x_k to x_{k+1}. An edge in the directed graph from node x_k to node x_{k+1} corresponding to control u_k is associated to an edge cost equal to $c_k(x_k, u_k)$. The terminal edges from a node x_N at level N to node t are associated with the terminal cost $c_N(x_N)$. Clearly a trajectory of the system corresponds to a path in the graph from a

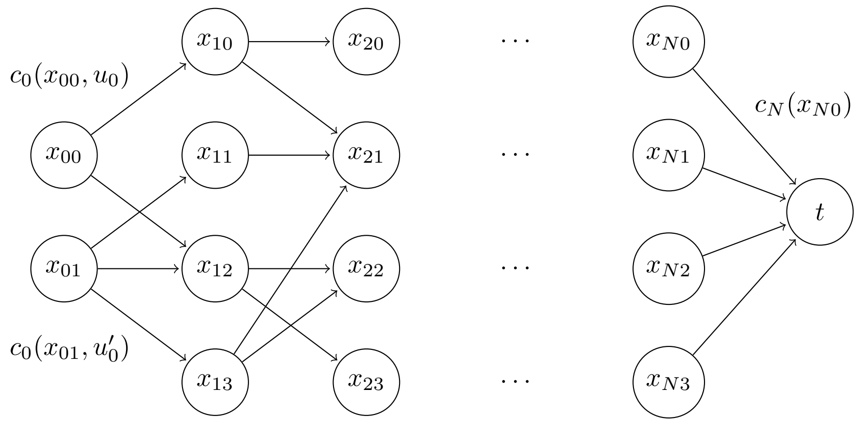

Figure 2.1: Directed graph representing a deterministic control problem. The nodes at stage k correspond to the possible values of x_k. Leaving a node x_k are edges corresponding to the possible controls $u_k \in \mathsf{U}_k(x_k)$. Two nodes x_k and x_{k+1} are linked if their state values satisfy $x_{k+1} = f_k(x_k, u_k)$. An edge from x_k corresponding to u_k has cost $c_k(x_k, u_k)$.

node at the initial level $k = 0$ to a node at the terminal level $k = N$. The cost of a path from a node a level $k = 0$ to the terminal node t is defined as the sum of the edges of the path. We see therefore that the cost (2.1) of a sequence of control $u_0, \dots, u_{N-1}$ starting from x_0, including the terminal cost, is equal to the cost of the corresponding path in the graph from x_0 to the terminal node t. *The optimal cost starting at x_0 is equal to the cost of the shortest path from x_0 to t.*

The DP algorithm for computing the optimum trajectory from x_0 to t is a direct translation of the algorithm (1.12), (1.13), presented in section 1.2. We can denote the cost of a transition from the state value $i \in \mathsf{X}_k$ at level k to state value $j \in \mathsf{X}_k$ at level $k+1$ by c^k_{ij} instead of $c_k(x_k, u_k)$ (the control u_k determines the resulting state j uniquely in a deterministic problem). Then the DP algorithm is

$$\begin{aligned}
&J^*_N(i) = c^N_{it}, \quad i \in \mathsf{X}_N, \\
&J^*_k(i) = \min_{j \in \mathsf{X}_{k+1}} \left[c^k_{ij} + J^*_{k+1}(j) \right], \quad i \in \mathsf{X}_k, k = 0, \dots, N-1.
\end{aligned}$$

The optimal cost $J^*_0(x_0)$ is then the cost of the minimum cost path from x_0 to t. The quantity $J^*_k(i)$ is the optimal cost-to-go from $i \in \mathsf{X}_k$ to t.

Conversely, consider a directed graph with N nodes labeled $1, 2, \dots, N, t$, and edge costs c_{ij} for edge (i, j). Suppose we want to find the shortest path from each node i to node t. Note that this problem requires that there is no cycle of negative cost in the graph, otherwise there are paths of cost $-\infty$. In this case, any shortest path uses at most N edges. We reformulate the problem to fit the set-up of the previous paragraph. Create a new graph with N levels

$k = 0, \dots, N-1$ and N nodes at each level, plus an additional level $k = N$ with only node t. Nodes at level k correspond to the N nodes of the original graph, and are connected only to nodes at level $k+1$. Node i at level k is connected to node j at the next level if there is an edge (i, j) in the original graph, and this edge is associated to a cost c_{ij}. If there is no edge (i, j) in the original graph, we still add an edge from i at level k to j at level $k+1$, but with cost $+\infty$. We also have all edges from i to i at successive levels, with associated cost 0, which means considering paths with degenerate moves (staying at the same node for one move) in the original graph. This way, we can capture paths of length less than N in the original graph. It is easy to see now that finding shortest paths in both graphs is equivalent. But the new graph is exactly the graph of an optimal control problem.

Algorithmic Implications

From the previous discussion, we can conclude the following. First, the problem of computing shortest paths in a graph with no negative cost cycle can be done using the dynamic programming algorithm. The discussion above leads to the DP recursion

$$J^*_{N-1}(i) = c_{it}, i = 1, \dots, N$$
$$J^*_k(i) = \min_{j=1,\dots,N} [c_{ij} + J^*_{k+1}(j)].$$

Here $J^*_k(i)$ has the intuitive meaning of the cost of the optimal path from j to t using $N-k-1$ moves ($J^*_k(i)$ will be $+\infty$ is no such path exists). The dynamic programming approach to solving the shortest-path problem is essentially the *Bellman-Ford* algorithm. There are other shortest-path algorithm which have better worst-case performance for certain types of problems (e.g., Djikstra's algorithm, which works only for edges with nonnegative cost), but DP is still very useful. The main issue with DP is that it explores every node of the graph, whereas for solving a single shortest path problem in a graph with a large number of nodes, most nodes are often not relevant for a particular shortest path. There are other methods that try to explore only the relevant part of the graph. A more detailed discussion of various shortest path algorithms can be found in [Ber07, setion 2.3] or any book on algorithm design [CLRS03].

Remark. Note that the optimal control problem obtained from the shortest path problem in a graph is time-homogeneous. That is, the edge cost c_{ij} does not depend on the level k of the node i. Hence, we see from the DP recursion that if $J^*_{k+1}(i) = J^*_k(i)$ for all i, then after that the DP iterations will not change the values of the cost-to-go any more so the algorithm can be immediately terminated.

Moreover, we see that by representing an deterministic finite-state finite-horizon optimal control problem in the form of a graph, *any shortest path algorithm* can be applied to solve the problem instead of the DP algorithm. There are indeed certain problems where other shortest path methods are preferable to DP.

The Forward DP Algorithm

In the case of deterministic problems, we have a DP algorithm that *progresses forward in-time*. Consider again Fig. 2.1. Now revert the direction of all the edges, keeping the cost values identical. Consider now a single node x_0 at level $k = 0$. An optimum trajectory from x_0 to t in the original graph is also an optimum path from t to x_0 in the "reversed graph". Let us call the optimum cost for the reversed problem $\tilde{J}_0(t)$, then

$$\tilde{J}_0^*(t) = J_0^*(x_0).$$

Now $\tilde{J}_0^*(t)$ can be computed by the DP algorithm on the reversed graph. This algorithm starts with the nodes at stage $k = 1$

$$\tilde{J}_N^*(j) = c^0_{x_0,j},\ \forall j \in \mathsf{X}_1.$$

And then proceeds forward on the graph

$$\tilde{J}_t^*(j) = \min_{i \in \mathsf{X}_{N-t}} \left[c_{ij}^{N-t} + \tilde{J}_{t+1}^*(i) \right], \quad j \in \mathsf{X}_{N-t+1},\ t = N-1, N-2, \ldots, 1.$$

The minimization above is now effectively over the states i at stage $N - t$ that can reach state j at stage $N - t + 1$. The optimal cost is then

$$\tilde{J}_0^*(t) = \min_{i \in \mathsf{X}_N} \left[c_{it}^N + \tilde{J}_1^*(i) \right].$$

Here $\tilde{J}_t^*(j)$ has the interpretation of the *optimum cost-to-arrive* to state j from state x_0, in $N - t + 1$ steps. Note that this forward DP algorithm is in fact the algorithm we used for the TSP at the beginning of this section.

Efficiency of the DP algorithm

From the graphical representation of Fig. 2.1, we can see why DP is an interesting algorithm from the computational point of view. For a given level k, the computations performed for $k = N, N-1, \ldots, k$ are all summarized in the value function $J_k^*(i), i = 1, \ldots, N$. Assume that at each level, there are at most n nodes. The computation of the optimum control value at each node using DP then requires a minimization over at most n numbers, hence n operations. Hence we see that computing $J^*(i)$ at each level requires n^2 operations. The total number of operations performed by the DP algorithm is therefore $O(Nn^2)$. This is much better than evaluating all trajectories through the graph, since there are $O(n^N)$ of these trajectories. Hence the DP algorithm is a fairly efficient algorithm if we can control the number n of nodes at each level, i.e., the number of values in the state-space $\mathsf{X_k}$. Unfortunatly, in most real-world problems, formulations tend to suffer from a state-space explosion problem, i.e., the state-space cardinality n typically grows exponentially with the problem parameters, as in the TSP example. This issue will occupy us for most of the second part of the course.

2.3 Application: Maximum-Likelihood Estimation in HMMs

We conclude this chapter with a classical algorithm for maximum likelihood estimation in Hidden Markov Models (HMM) [Rab89] that can be interpreted as a shortest path or DP algorithm. We consider a finite horizon N. An HMM is specified by a Markov chain $\{X_k\}_{1 \leq k \leq N}$, which we assume here homogeneous and with a finite number of states. We know the transition matrix $P(X_{k+1} = j | X_k = i) =: p_{ij}$ but we cannot observe the trajectories directly. Instead, we have access to observations $Y_k, k \geq 1$, that depend on the states X_{k-1}, X_k and are independent of the other variables X_j and observations Y_j given X_{k-1}, X_k. We are given the distributions $P(Y_k | X_{k-1} = i, X_k = j) =: r(Y_k; i, j)$ for all i, j. The control of such partially observable models will be the focus of chapter 5, but for now we only consider the following estimation problem. Given a sequence of outputs $y_{1:N}$, we wish to find the sequence of states $\hat{x}_{0:N}$ that is most likely, i.e., which maximizes $P(x_{0:N} | y_{1:N})$. Equivalently, since $P(x_{0:N} | y_{1:N}) = P(x_{0:N}, y_{1:N}) / P(y_{1:N})$ and $P(y_{1:N})$ is a fixed constant, we want to maximize $P(x_{0:N}, y_{1:N})$. It is not hard to see that

$$P(x_{0:N}, y_{1:N}) = \nu(x_0) \prod_{k=1}^{N} p_{x_{k-1}, x_k} r(y_k; x_{k-1}, x_k).$$

Maximizing this expression is equivalent to maximizing its logarithm

$$\max_{x_0, \ldots, x_N} \left\{ \ln(\nu(x_0)) + \sum_{k=1}^{N} \ln(p_{x_{k-1}, x_k} r(y_k; x_{k-1}, x_k)) \right\}.$$

The forward DP algorithm can be used to solve this maximization problem. It is better than the backward DP algorithm or shortest path algorithms when considering real-time applications, since we can execute the new step as soon as a new observation arrives. In this context, this algorithm is known as the *Viterbi algorithm*.

2.4 Practice Problems

Problem 2.4.1. Do all the exercises in the chapter.