2006-969: FRAMEWORK FOR DYNAMIC PROGRAMMING

Louis Plebani, Lehigh University
Dr. Plebani, P.E., is a faculty member in the Department of Industrial and Systems Engineering at Lehigh University where he teaches Dynamic Programming.

© American Society for Engineering Education, 2006

Framework for Dynamic Programming

Abstract

Dynamic programming (DP) is a versatile technique for modeling and solving sequential optimization problems. While the approach is well known to the Operations Research community, its impact has been limited when compared to other mathematical programming techniques such as linear programming. Ironically, in part, this has been due to its flexibility. Because DP can be adapted to a myriad of problems and those models can be implemented in a variety of ways, many modelers, particulary inexperienced ones, are overwhelmed by the large number of choices. This is often referred to as the "art" of dynamic programming. Our goal was to provide a framework and base computer code for students to achieve an ease of modeling and solution for dynamic programming similar to what has been achieved for linear programming. In so far as the teaching dynamic programming, this will allow educators in operations research to focus their teaching on issues relevant to dynamic programming as opposed to computer programming issues; and allow students in operations research to focus their learning on the power of dynamic programming, as opposed to the nuances of computer implementations.

Introduction

Since the formulation of Dynamic programming (DP) by Bellman,[1] it has been successfully applied to a variety of problems, including capacity planning, equipment replacement, production planning, production control, assembly line balancing and capital budgeting (hundreds of articles referring to the use of dynamic programming are given in Sniedovich and Cole[7]). Despite seemingly successful, dynamic programming has not been adapted nearly as readily, and thus successfully, as its mathematical programming counterparts such as linear and integer programming. Some of the reasons for this are the lack of standardization in representing dynamic programs and the lack of available software to aid implementation. Our work in developing a standardized framework is an attempt to address these shortcomings.

Dynamic programming can be defined as follows: At each *stage* in a process, a *decision* is made given the *state* of the system. Based on the state, decision and possibly the stage, a *reward* is received or a *cost* is incurred and the system *transforms* to another state where the process is repeated at the next stage. Note that this transformation can be either deterministic, where the resulting state is known with certainty, or stochastic, where a number of resulting states can occur with known probability. The idea of transforming or moving between states in time captures the concept of a sequential decision process as one evaluates decisions at each state in time and determines the optimal decision. The goal is to find the optimal *policy*, which is the best decision for each state of the system, or at least the optimal decision for the initial state of the system. It is the definition of a system according to states that makes dynamic programming so appealing, as it can model nearly any type of system.

Unfortunately, this flexibility generally requires some sophistication on the part of the user in model development. For example, a simple finite-horizon equipment replacement

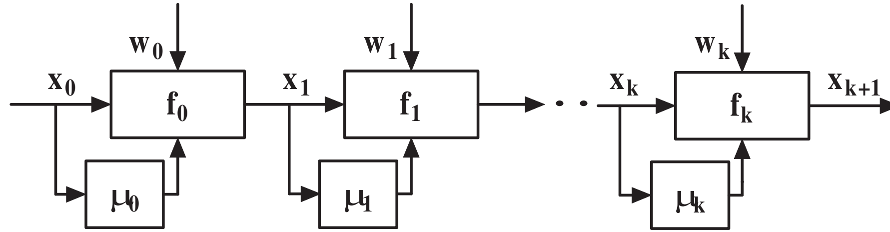

Figure 1: Basic Dynamic Programming Model

problem could be modeled using the asset age (Bellman[2]), the time period (Oakford et al.[5]), or the cumulative service (Hartman[4]) as the state of the system. Despite producing equivalent optimal policies, each model has different computational ramifications and data requirements. Sniedovich[6] notes similar issues when modeling knapsack problems. Sniedovich also notes that when examining a DP formulation, students often remark that it is "intuitive and obvious" but when asked to construct a formulation to a slightly different problem, they discover that the exercise is not trivial. This is why the use of dynamic programming is often referred to as an "art" (Dreyfus and Law[3]). An overarching goal of our framework is to reduce the art required to develop and solve a dynamic program.

Another issue that complications the teaching of DP is the fact that the user must be reasonably sophisticated with respect to implementation. This is immediately recognized by any instructor when teaching dynamic programming. Many students become frustrated when having to spend disproportionate amounts of time in writing and debugging programs to implement the DP algorithms they develop for assigned exercises. Our concern was not that the computer programming issues were not important, but that too much time was being cannibalized from DP modeling issues (such as choosing between the available equipment replacement models described above) in order to discuss computer science issues.

Specifically, the objectives of our developing the framework was to assist in the instruction of dynamic programming by:

1. Standardizing the Representation of Dynamic Programs: The versatility of dynamic programming is highlighted by the various ways in which one can "write a DP down." A systematic approach is devised for a user to define a DP in order to proceed towards a solution.

2. Developing Dynamic Programming Solver Software: Given a standard form of user input, the framework provides a solver for finite horizon problems which minimize discounted, expected costs.

Dynamic Programming Formalities

The generic dynamic programming model is shown in Figure 1. where the state x_k is an element of a space S_k, the control u_k is an element of a space C_k, and the random disturbance w_k is an element of a space D_k. The beauty of dynamic programming is that there are virtually no restrictions on the structure of the spaces S_k, C_k, and D_k.

One of the major divisions of DP models is whether the number of stages in finite or infinite. The most broadly applicable model is where a discrete dynamic system is optimally controlled over a finite horizon. The underlying dynamic system is described by the state equation

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

The random disturbance w_k is characterized by a probability distribution that may depend explicitly on x_k and u_k. The control $u_k = \mu_k(x_k)$ takes on values in a nonempty subset $U(x_k) \subset C_k$. A set of functions (control laws) $\pi = \{\mu_0, \ldots \mu_{N-1}\}$ defines a policy. The cost accumulates at each stage based upon the state, the control applied, and the value of the random disturbance. For known functions g_k, the expected cost of π starting at x_0 is

$$J_\pi(x_0) = E\left\{ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \right\}. \tag{2}$$

The optimal cost function J^* and optimal policy π^* satisfy

$$J^*(x_0) = J_{\pi^*}(x_0) = \min_{\pi \in \Pi} J_\pi(x_0). \tag{3}$$

In addition to finite horizon models, under assumptions that the system equation, the cost per stage equation, and the statistics of the random disturbance are stationary, DP has applicability for infinite horizon cases, i.e., cases where the number of stages is very large and for analysis purposes can be assumed infinite. In the these models, the optimal cost function most often takes the form of a discounted accumulated cost

$$\lim_{N\to\infty} \underset{\substack{w_k \\ k=0,1,\ldots}}{E} \left\{ \sum_{k=0}^{N-1} \alpha^k g(x_k, \mu_k(x_k), w_k) \right\}. \tag{4}$$

The techniques of dynamic programming are based upon the optimality principle as stated by Bellman:[1] "An optimal policy has the property that whatever the initial state and initial decision, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision". Direct application of the optimality principle to the finite problem of equation (3) yields the DP algorithm:

$$J_k(x_k) = \min_{u_k \in U_k(x_k)} \underset{w_k}{E} \left\{ \, g_k(w_k, u_k, w_k) + J_{k+1}\left(\, f_k(g_k, w_k, u_k, w_k) \, \right) \, \right\}. \tag{5}$$

where $J_k(x_k)$ is the optimal cost function (often called "cost-to-go") over stages $k, k+1, \ldots, N$ when the system is in state x_k at time k. Ideally, one would like to use (5) to obtain a closed-form solution for J_k. However, in most practical cases, an analytical solution is not feasible and numerical execution of the DP algorithm must be performed. Thus, the motivation for general software is clear.

The General Framework

We developed an object-oriented programming framework so that students studying dynamic programming are able to concentrate more on the study and development of DP

models and algorithms and spend less time coding and debugging the computer programs that implement them. This framework consists of a package of C++ template classes which together capture much of the implementation of the DP recursion (5). In order to develop a ready to run DP solver, the user: (1) determines the parameterized types involved in defining the state, control, and disturbance types; (2) implements necessary functions; and (3) instantiates the template classes. This results in a solver that is ready to compile and execute. A brief description of the classes in the DP package follows:

Objects of class `State` represent the state of the system (x_k). This class defines the mathematical and logical methods for state variables as required by the solver. In many problems this class can be satisfied by a simple type. In others, it may be necessary to represent a state as a more complex data structure, particulary where some of the assumptions of equation (1) are violated. One way to handle this is through state augmentation where the state space is enlarged. The `State` object is used in constructing virtually all template classes and functions in the package.

Objects of class `Control` represent the control (u_k) applied to a state k. The `Control` class encapsulates the mathematical and logical operations associated with `Control` objects. In many problems `Control` objects are a simple type. In more creative models, they can be more complex objects. The `Control` class is used in constructing many template classes and functions in the package.

Objects of class `Disturbance` supply the stage k disturbance (w_k) in stochastic problems. The constructor for this class takes `State` and `Control` objects as input values. When instantiated the `Disturbance` object provides a container type interface for access to the disturbance values and associated probabilities that are valid for the input `State` and `Control`.

`StateEquation` is a functor (function object class) that defines the state transition logic $x_{k+1} = f_k(x_k, u_k, w_k)$. Its constructor requires a state equation defined by the user as an input. Once instantiated, the `StateEquation` object is used to determine all state transitions.

The `Cost` class encapsulates the operations associated with cost-to-go values. Objects of this class are returned by the stage cost functions. In most cases it is a simple data type. It is used to specialize the template mathematical and logical operations pertaining to cost.

`ControlSet` set objects provide the set ($U_k(x_k)$) of `Control` objects that are valid for a particular stage and `State`. The constructor accepts a `State` object as an input argument. When instantiated it provides a container type interface for access to the `Control` objects that are valid for the current state.

`StageCost` is a functor that is used to determine the stage cost ($g_k(x_k, u_k, w_k)$) for supplied `State`, `Control`, and `Disturbance` objects. Its constructor takes a stage cost function and a final stage function defined by the user as an input. Once instantiated, this object is used to determine all values of stage cost.

`ConditionalCTG` is a functor used to calculate values of the conditional cost-to-go $J_k(x_k, u_k)$ for input `State` and `Control`. In stochastic models, it is responsible for performing expectation operations. In this case, it instantiates a `Disturbance` object and uses the supplied interface to calculate the required expectation. It has an option to store values for later retrieval if desired. The primary use for the storage option is to print reports for educational purposes or debugging.

`CostToGo` is responsible for storing the optimum cost-to-go values $(J_k(x_k))$ at each stage for feasible values of `State`. The associated optimum value of `Control` is also stored. When the accessor method is called for a particular stage and `State` value, it returns the optimum values if available or an indication that they need to be computed otherwise.

A `Solver` object directs the overall solution of the DP problem by instantiating and sending messages to the objects listed above. As implemented in the preliminary work, the solver directly applies the DP algorithm (5). The following steps provide a loose description of how the solver manages the objects in order to solve the DP:

1. A request is made to the `CostToGo` object for the values of the objects J_k and u_k corresponding to the value of the object x_k. (To "solve" the problem, a request would be made for the value of $J_0(x_0)$.)
2. If `CostToGo` has the requested value, i.e., it was previously computed and stored, it is returned. Otherwise, it is calculated using the following steps.
3. A value of `ControlSet` is instantiated that corresponds to the value of the x_k object. An appropriate iterator is returned and used to iterate over the control set and to request values of $J_k(x_k, u_k)$ from a `ConditionalCTG` object.
4. For a stochastic problem, the `ConditionalCostToGo` object instantiates a `Disturbance` object and computes the expectation:

$$\sum_{w \in W(x_k, u_k)} p(w) \left(g_k(x_k, u_k, w_k) + J_{k+1}(f(x_k, u_k, w)) \right)$$

   It is important to note that this will result in a reentrant request to the `Solver` object for the value of J^* corresponding to $x_{k+1} = f(x_k, u_k, w_k)$. While there is overhead associated with the recursive calls, this approach has the advantage of requiring calculations on results for state variable values that will be reached from earlier stages. This frees the user from having to determine reachable states in any stages a priori.
5. `CostToGo` uses the return values from `ConditionalCostToGo` to calculate

$$J_k(x_k) = \min_{u \in U_k(x_k)} J(x_k, u).$$

   The resultant value is stored so that subsequent requests for $J_k(u_k)$ are served immediately.

Experience

The initial framework was provided for student use in a course which involved fundamental finite horizon DP problems. The limited results were promising. The instructor was able to assign approximately twice as many homework problems as had previous be assigned in the course and the percentage of problems meaningfully completed by students increased by nearly a third.

The preliminary framework took a very straightforward approach to the implementation of the generic DP framework, particulary in terms of data storage and optimization. Standard library associative containers were used for storage and minimization was performed by iterating over a set and selecting the minimum value. The resulting robust framework satisfied the immediate goal of requiring minimal configuration and programming knowledge by the user. While adequate for introductory problems, in order for the generic framework to become a serious DP modeling framework for more advanced courses/problems having wide applicability, it must be expanded to provide more flexibility of data structures, memory allocation and solution algorithms.

In developing the preliminary framework, we relied on the template preprocessing capabilities of the C++ compiler/preprocessor. However, the framework did stress the capabilities of the template preprocessor. Some constructs had to be rewritten because they would return errors on some compilers and some constructs that we would have liked to have used were not supported at all. This situation is likely to be exacerbated as the implemented functionality becomes more complex. The effect of this is that while much of the generic nature of the framework can be supplied by the built-in template processing capabilities corresponding to the C++ specification (ISO/IEC 14882:2003(E)), manual text editing is likely to be required for the more complicated constructs. To greatly reduce, if not eliminate, the requirements of editing by the user, a DP meta or modeling language and associated preprocessor would likely need be created so that the non programming expert will be able to access the more advanced features by writing a meta language program. The output of the meta language program would be a C++ program which could be compiled with any C++ compiler supporting the current standard.

The `Solver` object in the preliminary framework executed a straightforward implementation of the DP algorithm of equation (5). This very general and robust solver is an adequate default solver, however, for more advanced work, several classes that users could selectively include in their model to modify or to replace the solver in order to adapt to the characteristics of the specific problem at hand should be provided. These additional features cold be grouped into areas that support: classic solver modifications; approximation techniques; and solving with the availability of a parallel machine or using a computational grid.

Conclusions

The use of a general framework can improve the teaching dynamic programming by allowing beginning students to experience the flexibility of dynamic programming in solving a myriad of problems from equipment replacement, resource allocation, and production planning to organ donor assignment and hospital bed capacity planning. Traditionally, the power of this flexibility could only be captured by users trained in both

dynamic programming (modeling issues) and computer science (implementation issues). This helps explain why dynamic programming is not utilized as prominently as it should be. The framework addresses the interface of modeling and implementation issues in order to make dynamic programming accessible.

Bibliography

1. R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.
2. R.E. Bellman. Equipment replacement policy. *Journal of the Society for the Industrial Applications of Mathematics*, 3:133–136, 1955.
3. S.E. Dreyfus and A.M. Law. *Art and Theory of Dynamic Programming*. Academic Press, New York, 1977.
4. J.C. Hartman and A. Murphy. Finite horizon equipment replacement analysis. *IIE Transactions*, to appear.
5. R.V. Oakford, J.R. Lohmann, and A. Salazar. A dynamic replacement economy decision model. *IIE Transactions*, 16:65–72, 1984.
6. M. Sniedovich. Dynamic programming revisited: Challenges and opportunities. Technical paper, Department of Mathematics and Statistics, University of Melbourne, Melbourne, Australia, 2005.
7. M. Sniedovich and G. Cole. Dynamic programming journal articles, 2005.