AC 2008-2623: USING A SCRIPTING LANGUAGE FOR DYNAMIC PROGRAMMING

Louis Plebani, Lehigh University

Using a Scripting Language for Dynamic Programming

Abstract

In this paper we present a simple programming framework that can be used in teaching Dynamic Programming at an introductory level. The target audience for the framework is the student in an introduction to dynamic programming setting who possesses rudimentary programming skills. The framework takes advantage of the late binding features of the Python scripting language to allow students to model their problem with somewhat arbitrary data types and and to subsequently solve without worrying about more complex computer programming issues.

Introduction

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.

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. 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. The problem is exacerbated when an instructor attempts to introduce dynamic programming into an introductory level course because many students in these courses have not achieved the programming skill level required to program the problem specific solvers in a reasonable time efficient manner. Inevitable student frustration leads to limited experience and a decrease in learning. The standardized framework contained herein 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 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 problem could be modeled using the asset age, the time period, or the cumulative service as the state of the system.[2,4] Despite producing equivalent optimal policies, each model has different computational ramifications and data requirements. Sniedovich[5] notes similar issues when modeling other type problems. He found 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 Dreyfus and Law[3] referred to the use of dynamic programming an "art". 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 (at least at the introductory level) by providing a flexible framework for the representation of dynamic programs and to provide students a solver for problems modeled in this framework.

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.

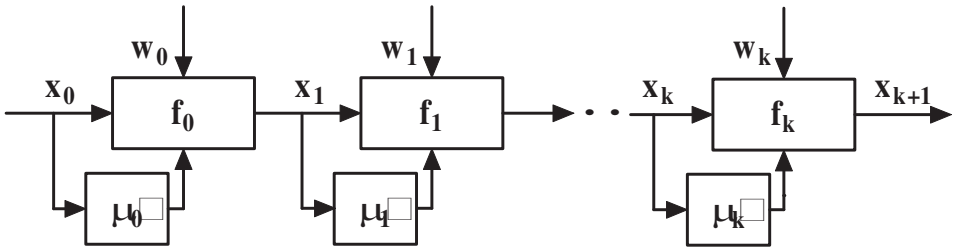

Figure 1: Basic Dynamic Programming Model

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

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

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

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

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:

$$J_k(x_k) = \min_{u_k \in U_k(x_k)} \underset{w_k}{E} \{ \, g_k(w_k, u_k, w_k) + J_{k+1} \, (\, f_k(g_k, w_k, u_k, w_k) \,) \, \} \, . \tag{4}$$

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 Equation (4) 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. This typically requires the student to develop a computer program "from scratch", which entails computer programming design issues such as appropriate data structures, data storage, and efficient program control. Students, particulary those taking introductory level courses, are often forced to spend disproportionate amounts of time getting their programs to work. They abandon their programs and learning suffers. Thus, the motivation for a simplified framework and general software.

The Framework

The framework consists of a module of Python classes which together capture much of the logic and implementation of the DP recursion in Equation (4). In order to develop a ready to run DP solver, the student: (1) determines the data types involved in defining the state, control, and disturbance; (2) implements necessary functions; and (3) instantiates the dynamic programming problem using the classes in the framework module. This results in a solver that is ready to compile and execute. A brief description of the elements and objects involved in using the framework to define and solve a dynamic programming problem follow.

Objects of class `State` represent the state of the system (x_k). The `State` object is used in instantiating nearly all objects inside the framework. Obviously, students are responsible for knowing the state variables of the system they are modeling and subsequently defining them. In more complex problems, i.e., problems with unusual representations or combinations of state variables, students would represent the state as a Python class. However, in most problems in an introductory course, the state can be represented as a simple type or Python tuple (combination) of simple types and no further effort by the student is required. In problems where a class is used, the student is also required to supply a hashing function for the class. The hashing function is used by Python dictionaries which is the mapping object used as the primary storage element type in the framework.

Objects of class `Control` represent the control (u_k) applied to a state k and is used in instantiating several objects inside the framework. For most problems in a introductory level course, `Control` objects are a simple type or combinations of simple types. In more creative models, they can be more complex objects. In this case, the student would define a Python class, in much the same manner as the `State` class, to represent the control.

Objects of class `Disturbance` supply the stage k disturbance (w_k) in stochastic problems. The constructor for this class takes the current stage k, state, and control to be applied 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`. Implementing the `Disturbance` would likely be the most involved programming a student would do. However, in many of the problems in an introductory course do not have a stochastic element and this class would not be required. If the class is required, the student has three major options. The first, and easiest, is to provide a function that would return a Python list of disturbance - probability pairs. The second, and probably as easy as the first, is to create a Python Generator. Generators are a simple and powerful Python tool for creating iterators. They are written like regular functions and essentially return objects that can be iterated over using the same python syntax as a list. The third possibility, slightly more complex that the other two, though still somewhat simple, is to implement a complete Python class that provides iterator functionality.

The student is required to define a `StateEquation` as a Python function that defines the

state transition logic $x_{k+1} = f_k(x_k, u_k, w_k)$. The inputs to the function are objects of type state, control, and disturbance, although the latter may be omitted for non-stochastic problems. It returns the new state.

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, particulary for introductory courses, the student would use a simple Python numerical data type to represent cost. In these cases, the `Cost` class can be ignored as all required functionality is provided by the simple type. In more complex cases, which would be unusual for an introductory course, a Python class would be defined to represent the `Cost` class. This class would be required to define the necessary operators for addition, comparison, etc., that would make objects of the class syntactically equivalent to the Python built-in numerical types. The stage cost and final cost class would then need to be defined in terms of the this class as defined below.

The student must define a function `ControlSet` that provides an object that contains the set $U_k(x_k)$ of `Control` objects that are valid for a particular stage k and state x_k. In most cases, the student would define a function that accepts the stage k and state x_k as an input arguments and returns a Python list of appropriate controls. Optionally, the student could provide a function with the same arguments that returns any Python object that implements the iterator interface. This second approach is unlikely to be required, especially in an introductory course.

The student must define a Python function to represent the cost at each stage $g_k(x_k, u_k, w_k)$. The function returns a object of type `Cost` for input arguments `State` u_k, `Control` u_k, and `Disturbance` w_k. The student must also define a Python function to represent the final cost $h(x)$. The function accepts the final state x as input and returns a object of type `Cost`.

The `CostToGo` object is responsible for storing the optimum cost-to-go values $J_k(x_k)$ for each stage for feasible values of x_k and is a critical part of the framework. It also stores the associated value of the control u_k that resulted in the optimal cost-to-go. `CostToGo` is an object internal to the framework and would not be accessed by the student using the framework. It is the late or dynamic binding of the Python scripting language that is exploited in this object and is a key element of the framework. There is no need to know the type of objects returned or used as indexes until they are defined by the student and the `CostToGo` object by the framework. When the `CostToGo` accessor method is called for a particular stage k and state x_k, it returns the optimum value if available, otherwise an indication is returned that the value needs to be computed.

The `ConditionalCTG` object is responsible for storing the optimum conditional cost-to-go $J_k(x_k, u_k)$ for each stage for feasible values of input state x_k and control u_k. Similar to the `CostToGo` object, it is an object internal to the framework and would not be accessed by the student using the framework. It also takes advantage of the dynamic binding nature of Python. When the `ConditionalCTG` accessor method is called for a particular stage, state,

and control, it returns the optimum value if available, otherwise an indication is returned that the value needs to be computed.

A `Solver` object directs the overall solution of the DP problem by instantiating and sending messages to the objects listed above. The solver directly applies the DP algorithm of Equation (4). 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. The `ControlSet` function is called with arguments k and state x_k. An iterable object is returns containing the feasible controls for state x_k. The returned object is used to iterate over the control set and to request corresponding values of $J_k(x_k, u_k)$ from the `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 experienced in this one course were promising. The instructor reported that he was able to assign approximately twice as many homework problems as he had previous in previous years and the percentage of problems

meaningfully completed by students increased by approximately a half. He also reported that classroom discussion centered around problem modeling instead of programming issues. Given that students study programming design issues in other courses, he could see no downside in using the framework.

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. R.V. Oakford, J.R. Lohmann, and A. Salazar. A dynamic replacement economy decision model. *IIE Transactions*, 16:65–72, 1984.
5. M. Sniedovich. Dynamic programming revisited: Challenges and opportunities. Technical paper, Department of Mathematics and Statistics, University of Melbourne, Melbourne, Australia, 2005.