Control and Cybernetics

vol. **35** (2006) No. 3

Dynamic Programming: an overview

by

Moshe Sniedovich[1] **and Art Lew**[2]

[1] Department of Mathematics and Statistics
The University of Melbourne, Australia

[2]Department of Information and Computer Sciences
University of Hawaii at Manoa, Hawaii, USA

e-mail: m.sniedovich@ms.unimelb.edu.au, artlew@hawaii.edu

Abstract: Dynamic programing is one of the major problem-solving methodologies in a number of disciplines such as operations research and computer science. It is also a very important and powerful tool of thought. But not all is well on the dynamic programming front. There is definitely lack of commercial software support and the situation in the classroom is not as good as it should be. In this paper we take a bird's view of dynamic programming so as to identify ways to make it more accessible to students, academics and practitioners alike.

Keywords: dynamic programming, principle of optimality, curse of dimensionality, successive approximation, push, pull.

1. Introduction

By all accounts dynamic programming (DP) is a major problem solving methodology and is indeed presented as such in a number of disciplines including operations research (OR) and computer science (CS). However, there are strong indications that it is not as popular among lecturers and practitioners in these disciplines as it should be.

The main objective of this paper is to reflect on the state of dynamic programming in OR and CS with a view to make it more accessible to lecturers, students and practitioners in these disciplines.

To accomplish this task we address a number of issues related to the state of DP as an academic discipline and a practical problem solving tool:

1. Methodology
   Great difficulties have been encountered over the past fifty years in attempting to encapsulate the methodology of DP in a user-friendly format. We explain why this state of affairs is not likely to change soon, if ever.

The analysis includes a summary of Bellman's approach – based on the *Principle of Optimality* – and the axiomatic approaches that became popular in the 1960s and aimed at putting Bellman's approach on a more rigorous foundation.

2. Algorithms
   We discuss the basic algorithmic aspects of DP and comment on *Dijkstra's Algorithm* for the shortest path problem as a representative of an important class of DP algorithms.
3. Curse of dimensionality
   We reflect on the role of the *Curse of Dimensionality* in the context of DP and remind ourselves of the distinction that should be made between the complexity of DP algorithms and the complexity of the problems these algorithms are designed to solve.
4. Approximations
   The need for efficient methods for generating good "approximations" of exact solutions to DP functional equations is an important aspect of DP. We mention how need is reflected in the content of this special volume.
5. Software support
   We lament on the lack of "general purpose" commercial software for DP and explain why a major breakthrough will be difficult to make.
6. Parallel processing
   Some DP functional equations are particularly suitable for parallel computing. We briefly discuss this important practical aspect of DP.
7. Petri nets
   Since many DP problems can be modeled as state transition systems and solved by finding shortest paths in associated directed graphs, we introduce a generalization, known as Petri nets, and discuss some connections between DP and Petri nets.
8. Teaching and learning
   Teaching DP has always been a problematic task. We explain why this is so and discuss strategies for tackling this difficult but potentially very rewarding task.
9. Myths and facts
   We address a number of myths that somehow over the years became an integral part of DP.
10. Opportunities and Challenges
    We remind ourselves that the state of DP offers plenty of opportunities as well as challenges.

We then briefly describe the content of the other papers in this special volume and indicate how they relate to the topics discussed in this overview.

To simplify the discussion we shall limit the review to *serial deterministic* processes and focus on the analysis and solution of *optimization problems.*

It is assumed throughout the discussion that the reader has some familiarity with DP.

2. Methodology

DP is definitely a "general purpose" problem solving tool with a distinct flavor. One of the manifestations of this fact is that it is not too difficult to describe in general, non-technical, jargon-free terms the inner logic of DP and how it actualy works.

For the purposes of this discussion it is instructive to examine the outline described in Fig. 1 that can be regarded as a sort of "recipe" for DP.

Figure 1. **Recipe for DP**

Step 1: *Embed* your original problem in a family of related problems.

Step 2: *Relate* the solutions to these problems to each other.

Step 3: *Solve* the relationship between the solutions to these problems.

Step 4: *Recover* a solution to the original problem from this relationship.

The relationship between the solutions to the related problems constructed in Step 2 typically takes the form of a *functional equation.* In some situations the functional equation is so simple that the distinction between the last two steps is blurred.

While the above recipe is informative, it definitely falls far short of being user-friendly in that it is not very constructive. Indeed, while it is a fact of life that at the completion of a successful DP investigation it is not difficult at all to identify the exact roles of these steps, the recipe itself does not give us much guidance as to how these steps should actually be implemented.

For example, the recipe does not tell us how exactly the original problem should be embedded in a family of related problems. By the same token, the recipe does not give us any hint as to how we should go about formulating the relationship between these problems. And it definitely does not tell us how we should solve the functional equation constructed in Step 2.

This is not surprising. In fact, this is precisely what you should expect from a recipe of this nature within the context of DP and why many scholars regard some aspects of DP as "art" rather than science (e.g. Dreyfus and Law, 1977). The main reason for this unavoidable ambiguity is the fact that the recipe applies to a vast class of disparate problems. The "art" of DP is all about being able to apply this "general" recipe in the context of *specific* problems.

It should be stressed, though, that this difficulty applies to the modeling aspects of other problem solving tools. In particular, many students have difficulties coping with the modeling aspects of, say, integer programming. But there are strong indications that the presence of this difficulty is especially felt in the area of DP.

We now consider two examples. The first is very simple, in fact trivial, the second is quite difficult.

EXAMPLE 2.1

The purpose of this example is to illustrate that the DP methodology is extremely pervasive. So much so that it is used everywhere by persons who know know nothing about DP itself.

Let x denote a sequence of numbers. Your task is to determine the sum of all the elements of x, call it $Sum(x)$. An obvious "solution" to this problem is to apply the definition of Sum and write:

$$Sum(x) = \sum_j x_j \tag{1}$$

assuming that the summation is carried out over all the elements of x.

The DP solution is a bit different, because according to our *Recipe* we need to go through four steps, the first of which requires us to embed our problem within a family of related problems. We do this by the following innocent looking definition:

$$Sum(y) := \sum_{j=1}^{n} y_j, \ y \in \mathbb{R}^n \tag{2}$$

where $\mathbb{R}$ denotes the real line.

Next, we have to relate the solutions to all these problems. This is not difficult because clearly the definition of Sum implies that

$$Sum(y) := Sum(y_1, \ldots, y_{n-1}) + y_n, \ y \in \Re^n \tag{3}$$

We can now apply this relationship to solve our original problem, namely we write

$$Sum(x) = Sum(x_1, \ldots, x_{n-1}) + x_n \tag{4}$$

where here n denotes the length of sequence x.

To recover a solution to our problem we just have to compute the value of $Sum(x)$ in accordance with (4). This can be done in various ways, e.g. recursively or non-recursively.

To highlight the basic structure of this "solution" to the problem under consideration, take a look at the pseudo-code in Fig. 2 for computing the sum of the elements of a list.

```
subroutine MySum(x,n);
    sum = 0;
    For j=1,...,n Do:
       sum = sum + x(j);
    End     return sum;
```

Figure 2. DP code for computing $\sum_{j=1}^{n} x_j$

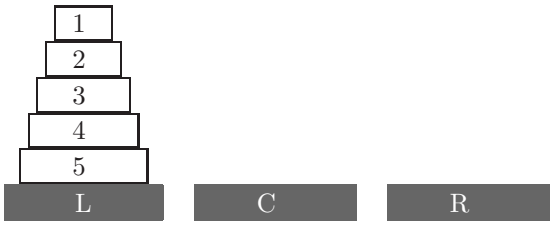


Figure 3. Tower of Hanoi Puzzle

The important thing to observe is that the process of computing the required sum also computes the sum of the sublists of the original list even though these subsums were not requested.

Example 2.2 (The Towers of Hanoi Puzzle)

Move the five pieces in Fig. 3 from the left platform to the right platform, one piece at a time. Make sure that at all times at each platform the labels of the pieces are arranged in ascending order (from top to bottom).

If you have not solved such puzzles before, you may find it exceedingly difficult to apply the DP Recipe in this context. Here is how it goes:

Step 1: The key to a successful application of this crucial step is the following rather simple[1] observation: The problem under consideration can be parameterized by three bits of information: (1) the number of pieces that we have to move (2) the present location of these pieces and (3) the destination of these pieces. So, we define the following family of puzzles

$P(n, x, y) :=$ A problem involving n pieces that must be moved from platform x to platform y.

In our case we can let n be an element of set $\{1, 2, 3, 4, 5\}$ and x and y be elements of $\{L, C, R\}$. Now let $S(n, x, y)$ denote the solution to $P(n, x, y)$, that is the sequence of moves that solves $P(n, x, y)$.

[1] Especially if you already know the answer!

Step 2: Because of the restrictions on how we can move the pieces, it is clear that in order to move the largest piece from platform x to platform y we must first move all other pieces from platform x to the third platform. This implies that for $n > 1$ we have

$$S(n, x, y) = S(n-1, x, !(x, y)) \cdot S(1, x, y) \cdot S(n-1, !(x, y), y) \qquad (5)$$

where $\cdot$ denotes concatenation and $!(a, b)$ denotes neither a nor b.

Step 3: There are various ways to solve (5). In particular, if n is not large it might be convenient to solve (5) recursively. Observe that $S(1, x, y)$ means that we move a piece from platform x to platform y, hence $S(1, x, y)$ is regarded as known: $S(1, x, y) :=$ *move a piece from platform x to platform y.*

Step 4: Our mission is to determine the value of $S(5, L, R)$. Any method that was identified in Step 3 can be used for this purpose. In particular, since n is small in our case, (5) can be easily solved recursively.

It should be pointed out that the DP formulation presented above is *non-serial* in nature: we express the solution to the subproblem of interest in terms of *two* other problems. It should also be indicated that although the functional equation does not include a *min* operation, the solution it generates is optimal: it minimizes the number of moves.

More details on the DP treatment of this famous puzzle, including interactive web-based modules, can be found in Sniedovich (2002).

These examples illustrate the idea that underlies the methodology of DP. Of great practical importance is that this basic idea can be extended to solve major classes of important problems. It is to the class of *optimization problems* that DP has been most applied in practice and therefore this review focuses on problems of this type.

Bellman's (1957) strategy for tackling the methodological aspects of DP was to keep things simple even at the expense of generality. Furthermore, he endeavoured to formulate the basic idea of DP as a *Principle*. He also deliberately decided to frame DP as a whole as a method for dealing with *sequential decision processes.* The end product is a methodology based on the following fundamental concepts:

- *Sequential Decision Process.*
- *Principle of Optimality.*
- *Functional Equation.*

Recall that the *Principle* was stated as follows (Bellman, 1957, p. 83):

> Principle of Optimality. An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with respect to the state resulting from the first decision.

The conceptual framework is then as follows: the process starts at the first stage $j = 1$ with some initial state $s_1 \in S$ where a decision $x_1 \in D(1, s_1)$ is made whereupon the process moves to the next stage $j = 2$ where the state $s_2 = T(1, s_1, x_1) \in S$ is observed. Then the second decision $x_2 \in D(2, s_2)$ is made whereupon the process moves to the next stage $j = 3$ where the state $S_3 = T(2, s_2, x_2) \in S$ is observed, and so on. The process terminates at stage $j = n$ where the final decision $x_n \in D(n, s_n)$ is made whereupon the process moves to the final stage $j = n+1$ where the final state $s_{n+1} = T(n, s_n, x_n) \in S$ is observed and a reward $r(s_{n+1})$ is incurred. Note that no decision is made at the final stage $n + 1$.

We refer to S as the *state space*, to T as the *transition function*, and to $D(j, s)$ as the set of *feasible decisions* pertaining to state s at stage j. Associated with this process define

$$f_j(s) := \quad \text{maximum reward that can be generated at the end of the process given that at stage } j \text{ we observe state } s.$$

Our mission is then to determine the value of $f(s_1)$ and to identify the optimal values of the decision variables $(x_1, \ldots, x_n)$.

THEOREM 2.1

$$f_{n+1}(s) = r(s), \ \ s \in S \tag{6}$$

$$f_j(s) = \max_{x \in D(j,s)} f_{j+1}(T(j, s, x)), \ \ j = 1, \ldots, n; s \in S \tag{7}$$

This is the *functional equation* of DP for the deterministic final state model formulated in Bellman (1957, pp. 82-83) as a framework for the introduction of the *Principle of Optimality.* Bellman viewed the functional equation as a mathematical transliteration of the *Principle,* and argued – correctly – that a proof (by contradiction) of the validity of the *Principle* – hence the functional equation – is immediate in this case.

In practice the total cost/return often depends on intermediate decisions and/or states, and the DP functional equation can cope with such cases. In fact, the most common objective function in practice is additive in nature, that is

$$g(s_1, x_1, x_2, x_3, \ldots, x_n) = \sum_{j=1}^{n} c(j, s_j, x_j) \tag{8}$$

where $c(j, s_j, x_j)$ denotes the cost/return generated at stage j by decision x_j given that the state is s_j and the corresponding DP functional equation is as follows:

$$f_j(s) = \max_{x \in D(j,s)} \big\{c(j, s, x) + f(j + 1, T(j, s, x))\big\}, j = 1, \ldots, n \ ; \ s \in S \tag{9}$$

with $f(N+1,s) = r(s), s \in S$.

As another example, here is a *stage-free* functional equation associated with an additive objective function in the framework of a *non-serial* process where after a decision is made the process diverges into two sub-processes:

$$f(s) = \max_{x \in D(s)} \left\{ c(s,x) + f(T_1(s,x)) + f(T_2(s,x)) \right\} \ , \ s \in S \backslash \{s'\} \tag{10}$$

with $f(s') = 0$.

Note that in response to applying decision $x \in D(s)$ to state s the system generates two new states, namely $T_1(s,x)$ and $T_2(s,x)$.

It is interesting to note that Bellman had no illusions about the difficulties associated with the application of the DP methodology in the context of specific problems (e.g. Bellman, 1957, p. 82):

> We have purposely left the description a little vague, since it is the spirit of the approach that is significant rather than the letter of some rigid formulation. It is extremely important to realize that one can neither axiomatize mathematical formulation nor legislate away ingenuity. In some problems the state variables and the transformations are forced on us; in others there is a choice in these matters and the analytic solution stands or falls upon this choice; in still others, the state variables and sometimes the transformations must be artificially constructed. Experience alone, combined with often trial and error, will yield suitable formulations of involved processes.

Needless to say, over the years DP scholars expanded the scope of operation of DP and developed numerous abstract axiomatic formulations of DP (e.g. Ellis, 1955; Brown and Strauch, 1965; Verdu and Poor, 1987; Bird and de Moor, 1997).

It is precisely for this reason that it is important to stress that Bellman's advice is as valid today as it was in 1957.

3. Algorithms

Generally speaking, there are two basic approaches to solving DP functional equations, namely via *direct* methods and *successive approximation* methods. The latter can be classified into two groups namely *pull*-type methods and *push*-type methods.

Direct methods solve the DP functional equation as "instructed" by the functional equation. For example, here is an outline of a direct method for solving (6)-(7).

Direct Method

Initialization:	Set $F_{n+1}(s) = r(s), s \in S$.
Iteration:	For $j = n, \ldots, 1$ - in this order - Do: $F_j(s) = \max_{x \in D(j,s)} F_{j+1}(T(j,s,x)), s \in S$.

Clearly, upon termination we have $F_j(s) = f_j(s), \forall j = 1, \ldots, n, s \in S$.

It should be noted that DP functional equations can often be solved *non-recursively*. When solved recursively, it might be necessary – for efficiency reasons – to adopt some means of avoiding recalculation of values, such as by saving them for reuse as needed (a process known as *memorization*). Often, an explicit stage variable is introduced so as to order the calculations with this in mind.

There are many situations, however, where direct methods cannot be used. Consider for example the following typical DP functional equation:

$$f(0) = 0 \tag{11}$$

$$f(s) = \max_{0 \le x \le s} \{\sqrt{x} + \beta f(s-x)\}, 0 \le s \le u \tag{12}$$

where $0 < \beta < 1$.

Applying the conventional method of *successive approximation*, we let F be an approximation of f and initialize F by setting $F^{(0)}(s) = 0, 0 \le s \le u$. We then update F repeatedly by the recipe

$$F^{(k+1)}(s) = \max_{0 \le x \le s} \{\sqrt{x} + \beta F^{(k)}(s-x)\}, k = 0, 1, 2, \ldots \tag{13}$$

It is not difficult to show (e.g. Sniedovich, 1992) that this updating procedure yields

$$F^k(s) = \sqrt{s\left[1 + \beta^2 + \beta^4 + \cdots + \beta^{2(k-1)}\right]} \tag{14}$$

so that at the limit as $k \to \infty$ the approximation becomes

$$F(s) = \sqrt{\frac{s}{1-\beta^2}} \tag{15}$$

which is the unique solution to the functional equation (11)-(12).

We call this type of updating mechanism *pull* because the update of $F(s)$ for a given s pulls the required $F(\cdot)$ values in accordance with the DP functional equation. Observe that the direct method outlined above is also based on a pull-type mechanism.

To describe the *pull* mechanism more formally, consider the following generic DP functional equation (e.g. Sniedovich, 1992):

$$g(s) = \max_{x \in D(s)} \{c(s,x) \oplus g(T(s,x))\} \tag{16}$$

where c and g are real valued functions and $\oplus$ is a binary operation. Then, formally

$$\text{Pull at } s: \qquad G(s) \leftarrow \max_{x \in D(s)} \{c(s,x) \oplus G(T(s,x))\} \tag{17}$$

where G denotes the approximation of g. In short, *pull* is an updating mechanism that mimics the DP functional equation in the context of which it is used.

The *push* mechanism, also known as *reaching* (e.g. Denardo, 2003), works a bit differently in that it pushes the value of $G(s)$ to states that can use it to update their current $G(\cdot)$ values. Thus, in the context of (16) we have

$$\text{Push at } s\text{:} \qquad G(s') \leftarrow \max\{G(s') \ , \ c(s',x) \oplus G(s)\} \tag{18}$$

where (s',x) is any pair such that $s' \in S, x \in D(s'), s = T(s',x)$.

The most famous DP successive approximation algorithm based on the *push* mechanism is no doubt *Dijkstra's Algorithm* (Dijkstra, 1959) for the shortest path problem. In this context the push operation can be restated as follows:

$$\text{Push at s:} \qquad v(x) \leftarrow \min\{v(x) \ , \ d(s,x) + v(s)\}, x \in Suc(s) \tag{19}$$

where $Suc(s)$ denotes the set of all *immediate successors* of node s, $d(s,x)$ denotes the length of $arc(s,x)$ and $v(s)$ denotes the approximated length of the shortest path from the origin to node s.

It is very unfortunate that the operations research and computer science literatures do not make this important aspect of *Dijkstra's Algorithm* clear. Sniedovich (2006) discusses this point at length. On the other hand, Lew (2006) shows how *Dijkstra's Algorithm* can be viewed as a greedy algorithm of a "canonical" type.

The successive approximation methods described above operate on the functional of the DP functional equation, e.g. on function g in (16) and function f in (12). That is, the functional of the DP functional equation is the object of the approximation scheme. In a similar manner it is sometimes possible and desirable to approximate the DP *policy*, that is the policy that determines the optimal values of the decision variables. The update mechanisms of such approximation schemes are therefore called *policy iterations* or *policy improvements* (e.g. Denardo, 2003) or *successive approximations in the policy space* (e.g. Sniedovich, 1992).

So as we have seen, a very important aspect of DP is the fact that the same functional equation can sometimes be solved by a variety of methods, hence algorithms. The situation is complicated even further because often the same problem can be formulated in a variety of ways, yielding a variety of functional equations. The following is a very famous case.

Example 3.1 (Unbounded knapsack problem)

Consider the following standard knapsack problem:

$$z^*(W) := \max_{x_1,\ldots,x_n} \sum_{j=1}^{n} v_j x_j \tag{20}$$

s.t.

$$\sum_{j=1}^{n} w_j x_j \leq W, x_j \in 0, 1, 2, \ldots \tag{21}$$

where W and $\{w_j\}$ are positive integers.

We shall consider two different DP functional equations for this problem, representing two different conceptual models of the problem.

Model 1: Suppose that the items are arranged in piles and we selected x_j items from pile j. Let $f_j(s)$ denote the maximum value of $v_j x_j + \cdots + v_n x_n$ subject to $w_j x_j + \cdots + w_n x_n \leq s$ and the integrality constraint. Then it is not difficult to show that

Theorem 3.1 *Let* $S := \{0, 1, \ldots, W\}$ *and* $J := \{1, \ldots, n\}$. *Then*

$$f_j(s) = \max_{\substack{x \leq s/w_j \\ x \in 0,1,\ldots}} \{xv_j + f_{j+1}(s - xw_j)\}, \forall s \in S, j \in J \tag{22}$$

where $f_{n+1}(s) = 0, \forall s \in S$.

This DP functional equation can be easily solved for $j = N, N-1, \ldots, 1$ - in this order provided that N and W are not too large. Note that this is an NP-hard problem and the time complexity of the algorithm based on a naive implementation of this DP functional equation is $O(KW^2)$ where $K := k_1 + \cdots + k_n$ and $k_j := 1/2w_j$.

Model 2: Suppose that we select the items from the piles one by one so that each time an item is selected the question is: from which pile should the next item be selected? Let $f(s) = z^*(s)$, namely let $f(s)$ denote the optimal value of the objective function in (20) when $W = s$. Then it is not difficult to show that

Theorem 3.2

$$f(s) = \begin{cases} 0 & , \quad s < \underline{w} := \min\{w_1, \ldots, w_n\} \\ \max\limits_{w_j \leq s} \{v_j + f(s - w_j)\} & , \quad s \geq \underline{w}. \end{cases} \tag{23}$$

The time complexity of the algorithm based on a naive implementation of this DP functional equation is $O(nW)$.

In summary, we have two very different DP models for the same problem. The DP functional equations induced by these models are substantially different and so is the complexity of the algorithms based on them, depending on the values of the parameters of the problem. It should be pointed out that other DP algorithms are available for this problem.

Even simple problems can be solved by different DP models; in fact, some problems can be solved by both a serial and a nonserial DP model. Some examples of this are given in Lew (2006).

4. Curse of dimensionality

This term refers to the phenomenon exhibited by many problems where the complexity of a problem increases sharply with its "size". It is interesting to note that this term was coined by Richard Bellman in his first book on dynamic programming (Bellman, 1957, p. xii).

In the context of dynamic programming this is usually manifested in a very large state space, namely in models where the state space is large and the DP functional equation is solved numerically.

For example, there are many problems, where there are more than 2^n distinct states where n is a parameter representing the "size" of the problem. For instance, in the case of the *travelling salesman problem (TSP)* there are $n2^n$ feasible states where n denotes the number of cities to be visited.

Of course DP is not the only methodology afflicted by this *Curse*. Indeed, complexity theory advises us that we should distinguish between the complexity of *problems* and complexity of *algorithms*. It should not surprise us therefore that the DP formulation of the TSP is subjected to the *Curse*, after all this problem is NP-hard.

On the positive side, it should be stressed that not all DP algorithms are cursed: there are many DP algorithms that are polynomial in time and space (e.g. DP algorithms for the shortest path problem). Furthermore, many DP functional equations can be solved *analytically* rather then numerically, in which case large state spaces do not cause any problem (e.g. (6)-(7)).

5. Approximations

In view of the above, it should not come as a surprise that attempts have been made to speed-up DP algorithms at the expense of the quality of the solutions they generate.

Of special interest are methods where the decision at each stage of a sequential decision process is made – deliberately – based upon incomplete information. *Greedy* or *myopic* algorithms are examples of such methods. Sniedovich (2006)

discusses *Dijkstra's Algorithm*, which is an example of a greedy algorithm that is optimal for nonnegative arc length, but approximate otherwise.

Other examples of optimal and approximate greedy algorithms are discussed by Lew (2006). An example of a myopic policy also appears in Piunovskiy (2006).

Other methods incorporate DP within a *heuristic* approach, where the objective is to obtain a "reasonably good" solution rather than an optimal solution.

The dire need for such compromises is reflected by the fact that a number of papers in this special volume (Hartman, 2006; Sniedovich and Voß, 2006; Wilbaut et al. 2006) present such methods. We discuss this in Section 12.

6. Software support

In view of the preceding analysis it is not surprising that at present there is no such thing as a "general purpose DP software package". This is in sharp contrast to say *linear programming* and *quadratic programming* where there are numerous commercial software packages capable of handling large problem instances. The situation is also much "better" in the area of *integer programming*.

What is currently available are specialized software packages addressing specific classes of problems. For example, there are well developed software packages for *Decision Tree Analysis* and *Critical Path Analysis*.

This state of the art means that practitioners often have to develop their own DP software. This could be a very rewarding but not necessarily easy endeavour. In fact, there are anecdotal evidence that this could be quite a tricky task. One reason for this is that the performance of some DP algorithms can be sped-up significantly using suitable *data structures*. Dijkstra's Algorithm is a good example (e.g. Denardo, 2003).

Despite the difficulties, efforts to develop better software support are ongoing. Lew and Mauch (2006) report on a promising approach based upon Petri nets. A brief review of relevant Petri net concepts is given below.

7. Parallel processing

One way to increase the efficiency and therefore practicality of DP is to consider possible use of parallel processing techniques to execute DP programs. This requires advances in both software and hardware technology. Appropriate software is needed to identify or detect portions of a program that can be executed in parallel, where each portion may be assigned to a separate processor in a distributed processing environment.

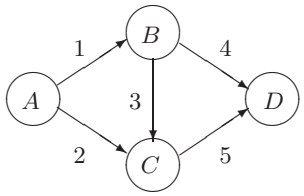
DP problems have many inherent sources of parallelism. For example, given a DP functional equation, independent parallel processors can be assigned to perform the calculations for each possible decision, after which the optimum of these calculated values can be determined.

In terms of software, greater efficiency can be achieved by the provision of suitable programming language features that enable users to explicitly identify parallellism in their programs, or of suitable compilers that automatically detect such parallelism. For example, modern programming languages such as java have multithreading features that programmers can use to indicate portions of their programs that can execute in parallel. Tereso, Mota and Lameiro (2006) discuss an application of this for a DP problem.

It is also possible to solve a given DP functional equation using a spreadsheet system, whose formula cells are used to compute the minima or maxima of other cells. Some cells would compute other minima or maxima and other cells would compute general functions of what are in other cells.

For example, a spreadsheet for finding the length of the shortest path in a graph (from Lew, 2000) is given in the following table:

	A	B	C	D
1	$= B1$	$= MIN(C1, C2)$	$= D1 + A2$	1
2	$= B2$	$= MIN(C3, C4)$	$= D2 + A3$	2
3	$= B3$	$= MIN(C5)$	$= D3 + A3$	3
4	0		$= D4 + A4$	4
5			$= D5 + A4$	5



Such DP spreadsheets are easy to produce for small problems, but not for large ones. Mauch (2006) discusses a software tool that can be used to automatically generate a spreadsheet that would solve a given DP functional equation.

In terms of hardware, a computer architecture where different portions of programs would execute on different processors in parallel would be advantageous. One such architecture is that of a dataflow computer in which processors are data-driven, where each processor awaits the arrival of data rather than the arrival of a command from a central processing control unit before it starts execution. For a DP spreadsheet, each formula cell would be assigned to a separate processor.

To formally model the solution of a DP functional equation, or the execution of a spreadsheet, or the behavior of a dataflow computer, it is useful to adopt a generalization of the usual state transition system model in which transitions are data-dependent. Petri nets, introduced in the next section, are one such model.

8. Petri nets

A Petri net – surveyed in Murata (1989) – is inherently a distributed or parallel processing system, hence a Petri net model of a DP problem permits it to be processed by a parallel system. Mauch (2006) describes a software tool that can be used to automatically generate a Petri net model of a DP problem.

DP can also be used to solve certain problems associated with Petri net models. As we note below, Werner (2006) uses DP to solve a critical path problem arising from a Petri net model, and Popova-Zeugmann (2006) uses DP to reduce the number of states associated with a Petri net model. We briefly review basic Petri net concepts below.

A Petri net is a class of directed graphs having two types of nodes, *place* nodes and *transition* nodes, where branches connect nodes of different types, and where place-nodes have ("contain") associated objects called *tokens*. For any transition-node T, branches from a place-node to the transition-node, or from the transition-node to a place-node, define the *input* place-nodes (or *preset*) and *output* place-nodes (or *postset*), respectively, of the given transition-node.

The state of a Petri net, called its marking M, is characterized by the tokens at each place-node P; the initial state of a Petri net is a specified initial marking $M0$. A Petri net makes transitions from one state to another based upon *firing rules* associated with the transition-nodes: if each input place-node of a transition-node contains a token, then the transition-node is eligible to fire.

The actual firing of a transition-node instantaneously causes "input" tokens to be removed from each place-node in its preset and "output" tokens to be inserted at each place-node in its postset. If more than one transition-node is eligible to fire, only one at a time may do so, chosen arbitrarily.

The *state transition diagram*, whose states are markings and whose state transitions correspond to transition-node firings, is called the reachability graph associated with the Petri net. This graph is a (possibly infinite) tree rooted at initial state $M0$.

Many properties of a Petri net can be determined by analysis of its reachability graph, such as the existence or optimality of certain paths. The latter suggests that DP can be applied to solve optimization problems related to Petri nets. Werner (2006) uses DP to solve a critical path problem arising from a Petri net model.

The foregoing basic definition of Petri nets has been extended in many ways. One class of extensions allows a token to have an attribute called its color, which may be a numerical value, and modifies the firing rules so as to require the colors of input tokens to satisfy certain conditions for eligibility and so as to compute a color to attach to each output token. Another class of extensions adds a time element to transitions in some way. One way is to assume the firing of a transition is no longer instantaneous, but may take a specified amount of time, possibly in a given range of values. Another way is to assume a transition may be eligible to fire only for a specified period of time, after which it becomes ineligible.

For these more complex Petri net models, the state of the Petri net must be redefined to incorporate additional information, hence the number of possible states will generally be much greater. Consequently, the equivalent reachability graph may be too large for analysis. Popova-Zeugmann (2006) uses DP to reduce the number of states associated with a Petri net model. Furthermore, of

interest is not only the use of DP to solve Petri net problems, but also the use of Petri nets to solve DP problems.

In such cases, a Petri net rather than a serial state-transition graph, is a more appropriate modeling and computational framework. Mauch (2006) describes a software tool based upon this idea that can be used to solve a very large class of DP problems; in essence, a Petri net model, called a *Bellman net* in Lew (2002), is adopted as an internal data structure representation for a given DP functional equation. This tool automatically constructs a *Bellman net* model of a DP functional equation, from which it is possible to automatically generate ordinary or parallel processing code to solve this equation.

9. Teaching and learning

Experience has shown that many students find DP to be extremely deceptive. When they examine a given DP formulation they regard it as intuitive and "obvious". Yet, when asked to construct a DP formulation to a slightly different problem, they suddenly discover that the exercise is not as easy as it first appeared to be. This is a manifestation of the difficulty known as the "art of dynamic programming". Dreyfus and Law (1977) have a very vivid discussion on this aspect of DP.

This is one of the reasons why teaching/learning DP can be a very rewarding experience but not necessarily an easy one.

Most students' first (and last!) encounter with DP is in introductory courses where DP constitutes only a small part of the content of the course. It is common practice in such environments to teach DP "by example". That is, there is not much discussion on the DP methodology. Rather, students are taught how to use DP to solve say 5-6 different types of problems. The hope is that through this experience they will somehow "get" the essence of this problem solving methodology.

It is not clear to what extent this strategy has been successful (see Dreyfus and Law, 1977, comments on this matter), but there seems to be very little that can be done in the short term to change this state of affairs. Lecturers practicing this strategy are advised that in this environment it is important to think carefully on a balanced collection of examples.

Since DP offers a versatile strategy for solving all sort of *games* and *puzzles*, it is only natural to consider one example illustrating this side of DP. The Towers of Hanoi Puzzle is a good choice (Sniedovich, 1992, 2002).

10. Myths and facts

Over the years a number of myths about dynamic programming have taken hold in the literature. We shall mention only four.

DP is a bottom-up technique

This myth is widespread. It creates the mistaken impression that the process of decomposing the original problem into subproblems necessarily yields smaller and smaller problems, until ultimately the subproblems become trivial. By implication this means that the solution procedure starts with " ... the smallest, and hence simplest, subinstances ... " (Brassard and Bartley, 1988, p. 142). This common view of DP completely distorts the fundamental idea of DP, which does not rely at all on the subproblems becoming "smaller" or "simpler". For example, in the case of shortest path problems with *cycles,* the subproblems do not become smaller nor simpler than the original problem. And in the case of *infinite horizon* problems, the subproblems are neither smaller nor easier than the original problem.

The role of decomposition in DP is first and foremost to create related problems and quantify the relationships between them and their solutions rather than to make the original problem smaller and smaller and/or easier and easier. Given that the method of *successive approximation* is so pervasive in DP, it is not clear why this myth is so prevalent.

Curse of dimensionality

As explained above, the *Curse* has to do with the size (cardinality) of the state space - not the dimension of the state variables. Therefore, it cannot be resolved merely by changing the representation of the state variables, as suggested for instance by Ram and Babu (1988). A short discussion on this misconception can be found in Sniedovich (1992, p. 184-185). On the positive side, it is important to stress that not all DP models are subject to the *Curse*. There are many situations (e.g. shortest path problems) where DP algorithms are efficient. Furthermore, certain instances of notoriously difficult problems are amenable to DP treatments. For example, as we already indicated above, for obvious reasons the DP formulation of the generic TSP is subject to the *Curse*. But this does not mean that all subclasses of this generic problem are difficult. For example, as shown by Balas and Simonetti (2001) certain subclasses on this generic problem can be solved by *linear time* DP algorithms.

The art of DP

It was suggested (e.g. Pollock and Smith, 1985) that the "art of DP" can be resolved by a simple enumeration-aggregation process where the state of the DP model is identified by inspection using the decision-tree representation of the underlying problem. This approach is fundamentally flawed. To start with, it requires the modeller to **enumerate all** the feasible solutions. Second, the states identified by this process are not necessarily the conventional DP states because the proposed method is prone to the whims of specialized circumstances associated with the particular instance of the problem under consideration. Thus, for

instance, the states generated by this method for an instance of the knapsack problem may not have the "physical" meaning of the conventional states associated with this problem ("capacity not yet utilized"). Thirdly, the approach cannot deal with *classes* of problems: how do you decide what is a proper state for the *generic* knapsack problem? In short, not only that *complete enumeration* cannot be accepted as a legitimate tool for determining the states of a DP model, the mere suggestion of this option is a clear indication of the enormous difficulties posed by the "art of DP".

Principle of optimality

This is no doubt the most controversial aspect of DP. Indeed, there are a number of myths regarding the meaning and validity of the *Principle* and its exact role in the theory and methodology of DP.

Given the long history of this controversy, it seems that it would be best for persons interested in this aspect of DP to consider the basic facts rather than continue the spread of speculations and unwarranted claims on this topic.

The basic issue here is not whether *Principle* is valid or invalid, but rather how it can be best used to explain what DP is all about (Sniedovich, 1992).

11. Opportunities and challenges

Our short discussion identified a number of problematic issues related to various aspects of DP. However, rather than regarding these issues as obstacles, we should view them as *challenges*, the point being that in each case there is certainly something constructive and useful that we can do about it.

In fact, this argument can be pushed even one step further, because where there is a challenge there can also be opportunities. Solving DP problems more efficiently is a major challenge that provides a variety of opportunities for researchers. Many of the papers in this Special Issue address this need for efficiency. Consider also the need for *user-friendly general-purpose DP software.* The lack of such software is a manifestation of the tremendous difficulties associated with the development of such products. But it also points out the obvious: the tremendous opportunities that exist in this very area.

This particular instance is also indicative of another aspect of DP, namely that the various challenges - hence opportunities - are interrelated to one another. Progress with the development of *user-friendly general-purpose DP software* will no doubt facilitate tackling the teaching/learning challenge.

Such software also contributes to greater efficiency since program development time (by people) is often more significant than program execution time (by computers).

12. Overview of papers in this volume

A number of approaches are available for making DP algorithm more efficient, naturally at the expense of not being able to guarantee the optimality of the solutions. We mention the following three:

1. Approximation techniques that obtain a tentative nonoptimal solution, often followed by techniques to improve these tentative solutions, or possibly a sequence of such approximate solutions that ideally converge to the exact optimal solution.
2. Heuristic techniques, where certain states or decisions can be excluded from consideration based upon, for example, greedy principles.
3. Parallel processing techniques, where advances in software or computer architecture are employed to solve the given equations.

Hartman and Perry (2006) describe the use of linear programming to obtain the approximate solution of a DP problem. Sniedovich and Voß (2006) describe a successive approximations techniques based upon search of neighborhoods defined by DP methods. Wilbaut et al. (2006) describe a successive approximations technique in which a tabu search method is used to improve partial solutions. Sniedovich (2006) characterizes *Dijkstra's algorithm* as a successive approximations technique.

Lew (2006) discusses classes of greedy algorithms that yield optimal solutions, including "canonical" greedy algorithms that can be derived directly from a DP formulation. One example of a canonical optimal greedy algorithm is Dijkstra's algorithm, as also discussed in Sniedovich (2006). Greedy or myopic policies used for Markovian decision processes are discussed in Piunovskiy (2006).

For a given DP formulation, efficiency can also be gained by software or hardware means. Tereso, Mota and Lamiero (2006) describe how a DP problem can be solved using distributed processing.

Mauch (2006) describes a software tool for DP that reduces program development time using a Petri net model. Finally, as mentioned above, Popova-Zeugmann (2006) and Werner (2006) discuss other problems related to Petri nets and DP.

13. Conclusions

The role and status of dynamic programming in such fields as operations research and computer science is well established, widely recognized and secured. There are strong indications, however, that its capabilities as a tool of thought and a versatile problem-solving methodology have not yet been fully utilized.

In this discussion we identified a number of challenges - hence opportunities - that must be dealt with in order to make dynamic programming more accessible to practitioners, researchers, lecturers and students.

Acknowledgement. This article is based on a tutorial and short paper entitled *Dynamic Programming Revisited: Opportunities and Challenges* (Sniedovich, 2004).

References

Balas, E. and Simonetti, N. (2001) Linear time dynamic programming algorithms for new classes of restricted TSP's: A computational study. *INFORMS Journal on Computing* **13** (1), 56–75.

Bellman, R. (1957) *Dynamic Programming.* Princeton University Press, Princeton, New York.

Bird, R. and de Moor, O. (1997) *Algebra of Programming.* Prentice-Hall, New York.

Brassard, G. and Bratley, P. (1988) *Algorithmics Theory and Practice.* Prentice-Hall, New York.

Brown, T.A. and Strauch, R.E. (1965) Dynamic programming in multiplicative lattices. *Journal of Mathematical Analysis and Applications* **12**, 364-370.

Denardo, D.E. (2003) *Dynamic Programming Models and Applications.* Dover, New York.

Dijkstra, E.W. (1959) A note on two problems in connexion with graphs. *Numerische Mathematik* **1**, 269-271.

Dreyfus, S.E. and Law, A.M. (1977) *The Art and Theory of Dynamic Programming.* Academic Press, New York.

Ellis, D. (1955) *An Abstract Setting of the Notion of Dynamic Programming, P-783.* The RAND Corporation, Santa Monica, CA.

Hartman, J.C. and Perry, T.C. (2006) Approximating the solution of a dynamic, stochastic multiple knapsack problem. *Control and Cybernetics* **35** (3), 535-550.

Lew, A. (2000) N degrees of separation: Influences of dynamic programming on computer science. *Journal of Mathematical Analysis and Applications* **249**, 232–242.

Lew, A. (2002) A Petri net model for discrete dynamic programming. *International Workshop on Uncertain Systems and Soft Computing*, Beijing.

Lew, A. (2006) Canonical greedy algorithms and dynamic programming. *Control and Cybernetics* **35** (3), 621-643.

Lew, A. and Mauch, H. (2006) *Dynamic Programming: A computational tool.* Springer, Berlin.

Mauch, H. (2006) DP2PN2Solver: A flexible dynamic programming solver software tool. *Control and Cybernetics* **35** (3), 687-702.

Murata, T. (1989) Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE* **77**, 541–580.

Pollock, S.M. and Smith, R.L. (1985) A formalism of dynamic programming. *Technical Report 85-8*. Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor, Michigan.

Piunovskiy, A.B. (2006) Dynamic programming in constrained Markov decision processes. *Control and Cybernetics* **35** (3), 645-660.

Popova-Zeugmann, L. (2006) Time Petri nets state space reduction using dynamic programming. *Control and Cybernetics* **35** (3), 721-748.

Ram, B. and Babu, A.J.G. (1988) Reduction of dimensionality in dynamic programming based solution methods for nonlinear integer programming. *International Journal for Mathematics and Mathematical Sciences* **11** (4), 811–814.

Sniedovich, M. (1992) *Dynamic Programming*. Marcel Dekker, New York.

Sniedovich, M. (2002) OR/MS Games: 2. The Towers of Hanoi Problem. *INFORMS Transactions on Education* **3** (1), 34–51.

Sniedovich, M. (2004) Dynamic Programming Revisited: Opportunities and Challenges, **41**, 1-11. In: Rubinov A. and M. Sniedovich, eds., *Proceedings of ICOTA 6*, December 9-11, University of Ballarat, Australia.

Sniedovich, M. (2006) Dijkstra's algorithm revisited: the dynamic programming connexion. *Control and Cybernetics* **35** (3), 599-620.

Sniedovich, M and Voss, S. (2006) The corridor method: a dynamic programming inspired metaheuristic. *Control and Cybernetics* **35** (3), 551-578.

Tereso, A.P., Mota, J.R.M. and Lameiro, R.J.T. (2006) Adaptive resource allocation to stochastic multimodal projects: a distributed platform implementation in Java. *Control and Cybernetics* **35** (3), 661-686.

Verdu S. and Poor H.V. (1987) Abstract dynamic programming models under commutative conditions. *SIAM Journal of Control and Optimization* **25** (4), 990–1006.

Werner, M. (2006) A timed Petri net framework to find optimal IRIS schedules. *Control and Cybernetics* **35** (3), 703-719.

Wilbaut, C., Hanafi, S., Freville, A. and Balev, S. (2006) Tabu search: global intensificaton using dynamic programming? *Control and Cybernetics* **35** (3), 579-598.