Control and Cybernetics

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

Dijkstra's algorithm revisited: the dynamic programming connexion

by

Moshe Sniedovich

Department of Mathematics and Statistics
The University of Melbourne, Australia
e-mail: m.sniedovich@ms.unimelb.edu.au

Abstract: Dijkstra's Algorithm is one of the most popular algorithms in computer science. It is also popular in operations research. It is generally viewed and presented as a greedy algorithm. In this paper we attempt to change this perception by providing a dynamic programming perspective on the algorithm. In particular, we are reminded that this famous algorithm is strongly inspired by Bellman's Principle of Optimality and that both conceptually and technically it constitutes a dynamic programming successive approximation procedure par excellence. One of the immediate implications of this perspective is that this popular algorithm can be incorporated in the dynamic programming syllabus and in turn dynamic programming should be (at least) alluded to in a proper exposition/teaching of the algorithm.

Keywords: Dijkstra's algorithm, dynamic programming, greedy algorithm, principle of optimality, successive approximation, operations research, computer science.

1. Introduction

In 1959 a three-page long paper entitled *A Note on Two Problems in Connexion with Graphs* was published in the journal *Numerische Mathematik.* In this paper Edsger W. Dijkstra — then a twenty nine years old computer scientist — proposed algorithms for the solution of two fundamental graph theoretic problems: the *minimum weight spanning tree problem* and the *shortest path problem.* We shall say a bit more about this paper later on in this discussion. For the time being suffice it to say that today Dijkstra's Algorithm (DA) for the shortest path problem is one of the most celebrated algorithms in computer science (CS) and a very popular algorithm in operations research (OR).

In the literature, this algorithm is often described as a *greedy algorithm.* For example, the book *Algorithmics* (Brassard and Bratley, 1988, 87-92) discusses it in the chapter entitled *Greedy Algorithms.* The *Encyclopedia of Operations Research and Management Science* (Gass and Harris, 1996, 166-167) describes

it as a "...node labeling greedy algorithm ..." and a greedy algorithm is described as "...a heuristic algorithm that at every step selects the best choice available at that step without regard to future consequences ..." (Gass and Harris, 1996, 264). And in Cormen et al. (1990, 519) we read:

> **Optimal substructure of a shortest path**
> Shortest-path algorithms typically exploit the property that a shortest path between two vertices contains other shortest paths within it. This optimal-substructure property is a hallmark of the applicability of both dynamic programming (Chapter 16) and the greedy method (Chapter 17). In fact, Dijkstra's Algorithm is a greedy algorithm, and the Floyd-Warshall algorithm, which finds shortest paths between all pairs of vertices (see Chapter 26), is a dynamic programming algorithm.

Although the algorithm is popular in the OR/MS literature, it is generally regarded as a "computer science method". Apparently, this is due to three factors: (a) its inventor was a computer scientist (b) its association with special data structures, and (c) there are competing OR/MS oriented algorithms for the shortest path problem. It is not surprising therefore that some well established OR/MS textbooks do not even mention this algorithm in their discussions on the shortest path problem (e.g. Daellenbach et al., 1983; Hillier and Lieberman, 1990) and those that do discuss it in that context present it in a stand-alone mode, that is, they do not relate it to standard OR/MS methods (e.g. Markland and Sweigart, 1987; Winston, 2004). For the same reason it is not surprising that the "Dijkstra's Algorithm" entry in the *Encyclopedia of Operations Research and Management Science* (Gass and Harris, 1996, 166-167) does not have any reference whatsoever to DP.

The objective of this paper is to present a completely different portrayal of DA, its origin, its role in OR/MS and CS, and its relationship to other OR/MS methods and techniques. Instead of focusing on its computational complexity, data structures that it can utilize to good effect, and other implementation issues, we examine the *fundamental principle* that inspired it in the first place, its basic structure, and its relationship to the well established OR/MS methods and techniques. That is, we show that at the outset DA was inspired by *Bellman's Principle of Optimality* and that, not surprisingly, technically it should be viewed as a *DP successive approximation procedure.*

Needless to say, this characterization of DA is not inconsistent with other valid characterizations. In particular, there is no conflict between this characterization and the common "greedy" characterization that is so popular in computer science circles. After all, DP offers a very natural paradigm for the formulation of greedy algorithms (Bird and de Moor, 1997, and Lew, 2005).

Be it as it may, one of the immediate implications of this fact is that OR/MS lecturers can present DA as an OR/MS-oriented algorithm par excellence and relate it to the well established OR/MS methods and techniques. In particular,

they can incorporate DA in the DP syllabus, where in fact it belongs. This, we should add, is long overdue for in teaching DA it is most constructive to draw the students' attention to the fact that this algorithm was inspired by *Bellman's Principle of Optimality*. Furthermore, it is important to point out that it is no more, no less, than a (clever) method for solving the DP functional equation for the shortest path problem given that cycles exist and the arc lengths are nonnegative.

The intimate relationship between DA and DP is definitely alluded to in the specialized literature (e.g. Lawler, 1976, and Denardo, 2003) but the details are not spelled out explicitly and are not easily accessible. An important objective of this discussion is to clarify these details and make them more accessible to computer scientists and operations researchers.

In short, our analysis aims at giving DA an OR/MS perspective by focusing on its relationship with standard DP methods and techniques. This, we hope, will contribute to a fuller understanding of both.

In the next section we describe the basic structure of the classical shortest path problem. We then describe DA and identify the class of shortest path problems it was designed to solve. This is followed by a detailed portrayal of DA as a DP algorithm. We then briefly discuss Dijkstra's (1959) famous paper and related matters.

Regarding content and style of presentation: we deliberately adopt a semi-formal approach and do not require significant DP background on the part of the reader. The discussion is readily amenable to a much less formal exposition and interpretation, as well as to a much more formal and rigorous technical treatment.

Since the topic under consideration should be of interest to lecturers teaching DP and/or DA as topics in specialized subjects and/or in much broader OR/MS and CS subjects, we provide an appendix containing discussions related to the "DP – DA" especially written with this kind of readers in mind.

2. Shortest path problems

One of the main reasons for the popularity of DA is that it is one of the most important and useful algorithms available for generating (exact) optimal solutions to a large class of *shortest path problems.* The point is that the shortest path problem is extremely important theoretically, practically, as well as educationally.

Indeed, it is safe to say that the shortest path problem is one of the most important generic problems in such fields as OR/MS, CS and artificial intelligence (AI). One of the reasons for this is that essentially any combinatorial optimization problem can be formulated as a shortest path problem. Thus, this class of problems is extremely large and includes numerous practical problems that have nothing to do with actual ("genuine") shortest path problems.

For the purposes of our discussion it is sufficient to consider only the "classical" version of the generic shortest path problem. There are many others.

Consider then the problem consisting of $n > 1$ cities $\{1, 2, \ldots, n\}$ and a matrix D representing the lengths of the direct links between the cities, so that $D(i, j)$ denotes the length of the *direct link* connecting city i to city j. This means that there is at most one direct link between any pair of cities. Formally we assume that $-\infty < D(i, j) \leq \infty, \forall i, j \in C := \{1, \ldots, n\}$ with $D(i, i) = \infty, \forall i \in C$. We interpret $D(i, j) = \infty$ as an indication that there is no direct link from city i to city j. It is useful to define

$$A(j) := \{i \in C : D(j, i) < \infty\} \ , \ j \in C \tag{1}$$

$$B(j) := \{i \in C : D(i, j) < \infty\} \ , \ j \in C \tag{2}$$

So, by construction $A(j)$ denotes the set of *immediate successors* of city j and $B(j)$ denotes the set of *immediate predecessors* of city j.

The objective is to find the shortest path from a given city, called *home,* to another given city called *destination.* The length of a path is assumed to be equal to the *sum* of the lengths of links between consecutive cities on the path. With no loss of generality we assume that city 1 is the home city and city n is the destination. So, the basic task is: **find the shortest path from city** 1 **to city** n**.**

For technical reasons, and with no loss of generality, it is convenient to assume that city 1 does not have any immediate predecessor. This is a mere formality because if this condition is not satisfied, we can simply introduce a dummy city and connect it to city 1 with a link of length 0. So we assume that $B(1) = \{\}$ where $\{\}$ denotes the empty set.

Note that in line with the above convention, should we conclude that the length of the shortest path from city i to city j is equal to ∞, the implication would be that there is no (feasible) path from node i to node j. It should also be pointed out that subject to the above conventions, an instance of the shortest path problem is uniquely specified by its distance matrix D. Thus, this matrix can be regarded as a complete model of the problem.

As far as optimal solutions (paths) are concerned, we have to distinguish between three basic situations:

- An optimal solution exists.
- There are no optimal solutions because there are no feasible solutions.
- There are no optimal solutions because the length of feasible paths from city 1 to city n is unbounded from below.

Ideally then, algorithms designed for the solution of shortest path problems should be capable of handling these three cases.

And in view of our interest in DA, we shall focus on shortest path problems where there might be cycles but the inter-city distances are non-negative. In this case we are assured that if a feasible solution exists then an optimal solution exists and that at least one of the optimal solutions is a *simple* path, namely it

is an *acyclic path.* Our objective is to construct such a path. In anticipation of this, let

$$f(j) := \text{length of shortest path from city 1 to city } j \in C. \tag{3}$$

Our objective is then to determine the value of $f(n)$. Observe that in view of the fact that $B(1) = \{\}$ it follows that $f(1) = 0$. Furthermore, if there is no feasible path from city 1 to city j then $f(j) = \infty$.

The fact that although our objective is to find the value of $f(n)$ we are interested also in the values of $f(j), j \neq n$, is a reflection of the very nature of both DA and DP: to compute the value of $f(n)$ we compute the $f(\cdot)$ values of other cities as well.

3. Dijkstra's algorithm

From a purely technical point of view DA is an iterative procedure that repeatedly attempts to improve an initial (upper) approximation $\{F(j)\}$ of the (exact) values of $\{f(j)\}$. The initial approximation is simply $F(1) = 0$ and $F(j) = \infty$ for $j = 2, \ldots, n$. In each iteration a new city is *processed:* its $F(\cdot)$ value is used to update the $F(\cdot)$ values of its immediate successors that have not yet been processed. A record, U, is kept of the set of yet to be processed cities, so initially $U = C = \{1, \ldots, n\}$. Upon termination the value of $F(n)$ is equal to $f(n)$. Full details are as follows:

Dijkstra's Algorithm: Version 1
$f(n) = ?$

Initialization:
$j = 1;\ F(1) = 0;\ F(i) = \infty,\ i \in \{2 \ldots, n\};\ U = C$
Iteration:
While ($j \neq n$ and $F(j) < \infty$) Do:
Update $U : U = U \backslash \{j\}$
Update $F : F(i) = \min\{F(i), F(j) + D(j,i)\}, i \in A(j) \cap U$
Update $j : j = \arg\min\{F(i) : i \in U\}$

Note that if termination occurs because $F(j) = \infty$, then $F(n) = f(n) = \infty$ regardless of whether or not $j = n$ at termination.

Sometimes it is required to determine the shortest distance from city 1 to every other city $j \in \{2, 3, \ldots, n\}$. To accomplish this task it is only necessary to modify the above formulation slightly as done below. At termination $F(j) = f(j), \forall j \in C$.

These two formulations of the algorithm do not explain how optimal paths are constructed. They only describe how the **length** of paths are updated. The

construction of optimal paths can be easily incorporated in these formulations by recording the *best immediate predecessor* of the city being processed. This predecessor is updated on the fly as $F(j)$ is being updated.

Dijkstra's Algorithm: Version 2
$f(j) = ?, j \in C$

Initialization:
$j = 1;\ F(1) = 0;\ F(i) = \infty, i \in \{2 \ldots, n\};\ U = C$
Iteration:
While ($|U| > 1$ and $F(j) < \infty$) Do:
Update $U : U = U \backslash \{j\}$
Update $F : F(i) = \min\{F(i), F(j) + D(j, i)\}, i \in A(j) \cap U$
Update $j : j = \arg\min\{F(i) : i \in U\}$

The main result is as follows:

THEOREM 3.1 *Dijkstra's Algorithm (Version 2) terminates after at most* $n - 1$ *iterations. If the inter-city distances are non-negative then upon termination* $F(j) = f(j),\ \forall j \in C$.

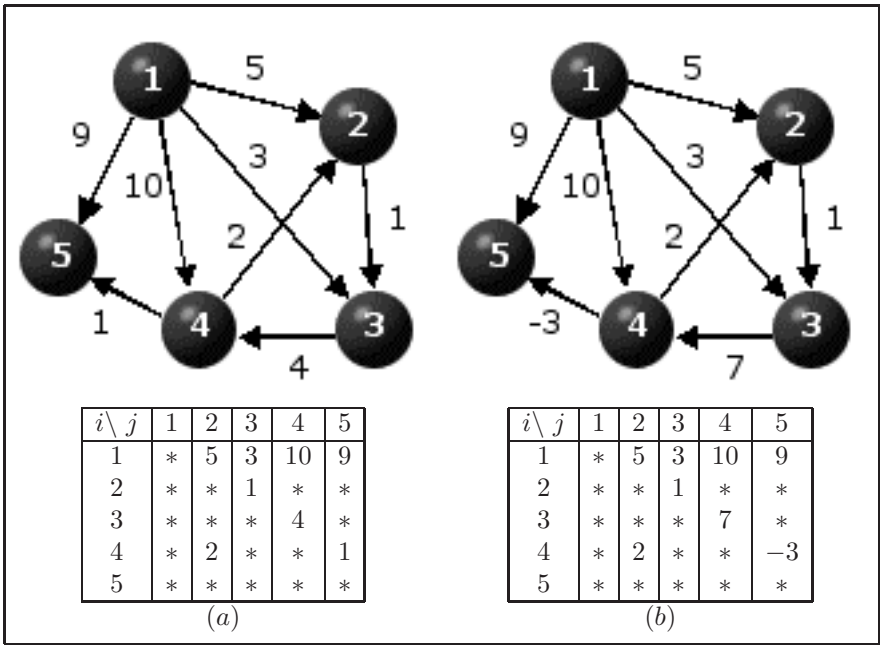
In short, if the inter-city distances are not negative the algorithms yield the desired results.

As far as computational complexity is concerned, DA performs rather well. At the m-th iteration we have $|U| := n - m$ for the *Update F* and *Update j* steps of the algorithm. For simplicity we assume that the min operation in the *Update j* step is conducted naively by pair-wise comparison. This means that at the m-th iteration the *Update j* step requires $n - m - 1$ comparisons. Table 1 summarizes the number of additions and comparisons conducted in the *Update F* and *Update j* steps of DA, where the superscript m refers to iteration m. Summing these values over $m = 1, \ldots, n - 1$ yields the total number of operations performed by the algorithm under the worst case scenario.

Table 1. Computational complexity of DA

	Additions	Comparisons
Update $F^{(m)}$	$n - m$	$n - m$
Update $j^{(m)}$	0	$n - m - 1$
Total ($m = 1, \ldots, n - 1$)	$n(n-1)/2$	$(n-1)^2$

So, in the worst case, the algorithm executes $n(n-1)/2$ additions and $(n-1)^2$ comparisons and the complexity of the algorithm is therefore $O(n^2)$. It should be noted, though, that special data structures (e.g. *heaps* and *buckets*) can be used to significantly speed up the arg min operation conducted by the *Update*



$i \backslash j$	1	2	3	4	5
1	$*$	5	3	10	9
2	$*$	$*$	1	$*$	$*$
3	$*$	$*$	$*$	4	$*$
4	$*$	2	$*$	$*$	1
5	$*$	$*$	$*$	$*$	$*$

(a)

$i \backslash j$	1	2	3	4	5
1	$*$	5	3	10	9
2	$*$	$*$	1	$*$	$*$
3	$*$	$*$	$*$	7	$*$
4	$*$	2	$*$	$*$	-3
5	$*$	$*$	$*$	$*$	$*$

(b)

Figure 1. Shortest path problems.

j step of the algorithm (Denardo and Fox, 1979; Gallo and Pallottino, 1988; Bertsekas, 1991; Ahuja et al., 1993; Denardo, 2003).

One of the important properties of DA is that if the distances are non-negative then the cities are processed in non-decreasing order of their $\{f(\cdot)\}$ values. That is, if $f(j) < f(i)$, then city j is processed prior to city i and consequently $F(j) = f(j)$ is confirmed prior to $F(i) = f(i)$. This means that if all the cities are being processed and are being relabeled as they are being processed so that the city processed at iteration m is labeled m, then upon termination $F(1) \leq F(2) \leq, \ldots, \leq F(n)$.

The following example illustrates the algorithm in action and highlights the fact that if the distances are allowed to be negative the algorithm may generate a non-optimal solution.

Example 1

Table 2 displays the results generated by DA (Version 2) for the two problems depicted in Fig. 1. The values in the j, U, F columns are the values of the respective objects at the **end** of the respective iteration. Iteration 0 represents the initialization step.

Table 2. Results generated by DA.

Iter	U	F	j
0	$\{1,2,3,4,5\}$	$\{0,\infty,\infty,\infty,\infty\}$	1
1	$\{2,3,4,5\}$	$\{0,5,3,10,9\}$	3
2	$\{2,4,5\}$	$\{0,5,3,7,9\}$	2
3	$\{4,5\}$	$\{0,5,3,7,9\}$	4
4	$\{5\}$	$\{0,5,3,7,8\}$	5

(a)

Iter	U	F	j
0	$\{1,2,3,4,5\}$	$\{0,\infty,\infty,\infty,\infty\}$	1
1	$\{2,3,4,5\}$	$\{0,5,3,10,9\}$	3
2	$\{2,4,5\}$	$\{0,5,3,10,9\}$	2
3	$\{4,5\}$	$\{0,5,3,10,9\}$	5
4	$\{4\}$	$\{0,5,3,10,9\}$	4

(b)

Note that in case (b) the algorithm failed to generate an optimal solution: $F(n) = F(5) = 9 > f(n) = f(5) = 7$.

4. DP perspective on the shortest path problem

To describe how DP approaches the solution of the shortest path problems we have to consider two distinct but obviously related things, namely

- DP functional equations for the shortest path problem
- Algorithms for solving DP functional equations.

For the purposes of our discussion it is convenient to regard f in (3) as a function on C, recalling that by definition $f(j)$ denotes the length of the shortest distance from node 1 to node j. This function has the following fundamental property:

THEOREM 4.1 (DP Functional equation for f defined in (3))

$$f(j) = \min_{i \in B(j)} \{f(i) + D(i,j)\} \ , \ j \in C \backslash \{1\} \tag{4}$$

observing that because we assume that $B(1) = \{\}$ *it follows that* $f(1) = 0$.

It should be stressed, here and elsewhere, that the DP functional equation **does not constitute an algorithm.** It merely stipulates a certain property that function f defined in (3) must satisfy. Indeed, in the context of Theorem 2 the functional equation constitutes a necessary optimality condition. This point is sometime not appreciated by students in their first encounter with DP.

Apparently this is a reflection of the fact that in many 'textbook examples' the description of the algorithm used to solve the functional equation is almost a carbon copy of the equation itself.

The question is then: how do we solve DP functional equations such as (4)? What kind of algorithms are available for this purpose?

So, recall that broadly speaking, DP deploys two types of methods for the solution of its functional equation, namely:

– Direct methods
– Successive approximation methods.

Before we discuss each of these two approaches in some detail, here is a very broad view of the scene.

Direct methods are based on a "direct" conversion of the functional equation of DP into an iterative procedure. Their scope of operation is therefore restricted to situations where it is possible to either solve the functional equation *recursively* or iteratively (non-recursively) in such a manner that while computing the value of $f(j)$ all the relevant values of $f(i)$ on the right hand side of the functional equation have already been computed (somehow). Since a truly recursive implementation is limited to relatively small problems, it follows that for most practical cases the direct approach is limited to cases where $C = \{1, \ldots, n\}$, the domain of function f, can be ordered so that for each $j \in C$ we have $i > j, \forall i \in B(j)$. This requires the shortest path problem to be *acyclic.* Conversely, if the problem is acyclic, the cities can be ordered in such a manner.

Not surprisingly, the introductory general purpose OR/MS textbooks deal almost exclusively with such methods and use examples where it is easy to arrange the elements of C in a proper order to enable the solution of the functional equation with the aid of direct methods. Perhaps one of the most well known applications of this approach is in the context of the Critical Path Method (CPM) where DP is used to compute early and late event times which are in fact specialized acyclic **longest path** problems. **Remark:** many OR/MS textbooks do not credit DP for this important application!

In contrast, successive approximation methods do not determine the value of $f(j)$ for a particular value of j by a single application of right-hand side of the functional equation. Rather, an initial approximation for $\{f(j) : j \in C\}$ is constructed, and then it is successively updated (improved) either using the functional equation itself or an equation related to it. The process terminates when a fixed point is reached, namely, when the incumbent approximation of f cannot be improved by the method.

There are a number of successive approximation methods. They differ in the mechanism they use to update the approximated $\{f(j)\}$ values and in the order in which these updates are conducted.

In this framework DA is viewed as a successive approximation method characterized by two properties. One is related to the mechanism used to update the approximated values of $\{f(j)\}$ and one to the order in which these values are updated.

And now the details.

4.1. Direct methods

Methods of this type are 'direct' applications of the DP functional equation. In the context of the shortest path problem they are used when the problem is *acyclic* namely when the cities can be labeled so that $D(i,j)=\infty, \forall i,j \in C, i \geq j$.

Generic Direct Method
$D(i,j)=\infty, \forall i,j \in C, i \geq j$

Initialization: $F(1)=0$
Iteration: For $j=2,\ldots,n$ Do:
$$F(j)=\min_{i\in B(j)}\{F(i)+D(i,j)\}$$

Clearly, upon termination the $\{F(j)\}$ values satisfy the DP functional equation (4) and $F(j)=f(j)$ for all $j \in C$. A major limitation of this method is that it cannot be used in cases where there are *cyclic* paths.

Observe that if the inter-city distances are non-negative, then in computing the value of $f(j)$ using (4) it is not necessary to consider any city i for which $f(i)>f(j)$. Hence,

LEMMA 4.1 *If* $D(i,j) \geq 0, \forall i,j \in C$ *then,*

$$f(j)=\min_{i\in B(j), f(i)\leq f(j)}\{f(i)+D(i,j)\}, j \in C\backslash\{1\}. \tag{5}$$

This suggests a direct method where the functional equation is solved in an order determined by the $f(\cdot)$ values rather than by the precedence constraints.

Modified Generic Direct Method
Assumption: $f(1) \leq f(2) \leq \cdots \leq f(n)$

Initialization: $F(1)=0$
Iteration: For $j=2,\ldots,n$ Do:
$$F(j)=\min_{i\in B(j), f(i)\leq f(j)}\{F(i)+D(i,j)\}$$

Clearly, upon termination the $\{F(j)\}$ values satisfy the DP functional equation (5) and $F(j)=f(j), \forall j \in C$.

Needless to say, this procedure is not very practical because the values of $\{f(j)\}$ are precisely the values we attempt to compute, hence they are not known a priori and therefore a priori it is not possible to label the cities so that $f(1) \leq f(2) \leq \cdots \leq f(n)$. Nevertheless, this procedure is instructive in that it suggests that the solution of the DP functional equation can be carried out not in a topological order (dictated by the precedence relation), but rather

in an order induced by the values of $\{f(j)\}$. This, in turn, suggests that it might be useful to solve the functional equation in an order determined **on the fly** (as the functional equation is being solved) rather than in an order that is pre-determined by the precedence relation.

This, then, suggests the deployment of a *successive approximation* approach for the solution of the functional equation that will generate (determine) the $\{f(j)\}$ values in a non-decreasing order. Not surprisingly, this is precisely how DA works.

4.2. Successive approximation methods

Successive approximation is a generic approach for solving equations and it is used extensively in DP, especially in the context of *infinite horizon* sequential decision problems (Bellman, 1957; Sniedovich, 1992; Denardo, 2003).

In a nutshell, these methods work as follows: an initial approximation to the unknown object of interest (e.g. function) is constructed. This approximation is successively updated (hopefully improved) until no further improvement is possible (or until it is determined that the process diverges rather then converges). The various methods differ in the way the updating is carried out and/or the order in which the updating mechanism is conducted.

In the case of the DP functional equation (4) we regard f defined in (3) as an unknown function on C and approximate it by some function F. We then successively update F using either the DP functional equation itself or an equation related to it. The updating process is repeated until a solution (fixed point) is found, namely until an approximation F is found that solves the equation governing the updating process.

There are two basic updating "philosophies". To describe the basic ideas behind them and the differences and similarities between them, assume that we currently have a new approximation F for f and that some of the new values of $\{F(\cdot)\}$ are smaller than the respective previous values. The basic question is then: how do we propagate the improvement in some of the $\{F(\cdot)\}$ values to other cities?

One approach to tackle this task, call it **Pull**, is inspired by the Generic Direct Method. It uses the functional equation (4) itself to compute new (hopefully improved) values of $\{F(j)\}$. That is, the basic update operation is as follows:

$$\textbf{Pull at j :}\ \ F(j) = \min_{i \in B(j)} \{F(i) + D(i,j)\}\ ,\ \ j \in C, B(j) \neq \{\}. \tag{6}$$

The term *Pull* is used here to convey the idea that when we process city j according to (4) we have to pull the most recently available values of $\{F(i) : i \in B(j)\}$. This operation is used extensively by DP in the context of infinite horizon problems in general (Bellman, 1957; Sniedovich, 1992) and Markovian decision processes in particular (Denardo, 2003).

The other approach to updating F, call it **Push**, propagates an improvement in $F(j)$ to the immediate successors of city j. That is, Push capitalizes on the fact that

$$f(j) = f(i) + D(i,j)\ ,\ j \in C, B(j) \neq \{\} \tag{7}$$

for some $i \in B(j)$. This fundamental property of the $\{f(\cdot)\}$ values is an implication of the *Principle of Optimality*, which we shall discuss in due course. This suggests the following simple updating recipe for F:

$$\textbf{Push at j :}\ F(i) = \min\{F(i), F(j) + D(j,i)\},\ i \in A(j). \tag{8}$$

Note that if $D(j,i) = \infty$, a Push at j will not improve the current value of $F(i)$. This is why a Push at j updates only the $\{F(i)\}$ values for $i \in A(j)$.

The term *Push* conveys the idea that if we observe an improvement (decrease) in the value of $F(j)$ then it makes sense to propagate (push) it to its immediate successors in accordance with (8). The improved value of $F(j)$ is thus pushed to the immediate successors of j so that they can check whether this improvement can actually improve their respective $\{F(\cdot)\}$ values.

The Push operation is commonly used by the DP methods such as *Reaching* and *Label-correcting* methods (Denardo and Fox, 1979; Bertsekas, 1991; Denardo, 2003).

The main difference between the Pull and Push operations is that a Pull at j attempt to improve the current value of $F(j)$ whereas a Push at j attempts to improve the current $\{F(\cdot)\}$ values of the immediate successors of j. In either case, to initialize the successive approximation process we need an *initial approximation* F for f. Ideally, this approximation should be easy to compute, guarantee that subsequent approximations of F will correspond to feasible solutions to the underlying problem and be non-increasing. The approximation $F(1) = 0$ and $F(j) = \infty, j \in C\backslash\{1\}$, can be used for this purpose. This is not a very exciting approximation, but it does its job.

A critical issue in the implementation of successive approximation methods is the order in which the states are processed. Let Q denote the set of cities queued for processing and let Select_From(Q) denote the element of Q selected to be processed next. The basic structure of the two successive approximation strategies is outlined in Table 3.

Observe that a single Pull operation updates a single $F(j)$ value, that is – a Pull at j updates only the value of $F(j)$. In contrast, a single Push operation typically updates several $F(\cdot)$ values, that is – a Push at j updates the values of $\{F(i) : i \in A(j)\}$. The difference and similarities between the Pull and Push "philosophies" are summarized in Table 4.

Table 3. Pull and Push

Successive Approximation for $f(j) = \min_{i \in B(j)} \{f(i) + D(i,j)\}, j \in C \backslash \{1\}; f(1) = 0$	
Pull	Push

Initialize:
$Q = A(1)$
$F(1) = 0; F(i) = \infty, i \in C \backslash \{1\}$
Iterate:
While ($\|Q\| > 0$) Do:
$j =$ Select_From(Q)
$Q = Q \backslash \{j\}$
$F(j) = \min_{i \in B(j)} \{F(i) + D(i,j)\}$
$Q = Q \bigcup A(j)$
End Do
	Initialize:
	$Q = \{1\}$
	$F(1) = 0; F(i) = \infty, i \in C \backslash \{1\}$
	Iterate:
	While ($\|Q\| > 0$) Do:
	$j =$ Select_From(Q)
	$Q = Q \backslash \{j\}$
	for $i \in A(j)$ Do:
	$G = F(j) + D(j,i)$
	if$(G < F(i))Do:$
	$F(i) = G$
	$Q = Q \bigcup \{i\}$
	End Do
	End Do
	End Do

Table 4. Pull vs. Push

Pull at j	Push at j
When city j is processed, the value of $F(j)$ is updated. If there is an improvement (decrease) in $F(j)$, then the immediate successors of j, namely the cities in $A(j)$, are appended to the list of cities to be processed ($Q = Q \bigcup A(j)$).	When city j is processed, the $F(\cdot)$ values of its immediate successors, namely $i \in A(j)$, are updated. If there is an improvement (decrease) in $F(i)$, then city i is appended to the list of cities to be processed ($Q = Q \bigcup \{i\}$).

Remarks

- For each $j \in C$ the Generic Direct Method computes the value of $F(j)$ exactly once. The two Successive Approximation methods typically compute (update) the value of $F(j)$ several times for a given j.
- Upon termination, the function F generated by the Successive Approximation procedures satisfies the DP functional equation (4), that is

$$F(j) = \min_{i \in B(j)} \{F(i) + D(i,j)\}, \forall j \in C \backslash \{1\}. \tag{9}$$

Thus, if the functional equation has a unique solution, the solution generated by the Successive Approximation procedure is optimal, that is $F(j) = f(j), \forall j \in C$.

- If the shortest path problem does not possess cyclic paths whose lengths are negative, then the Successive Approximation procedures terminate after finitely many iterations. In particular, termination is guaranteed if the distances are non-negative.
- If the problem possesses cyclic paths whose lengths are negative, the Successive Approximation procedures may not terminate. In fact, this property is commonly used to determine the existence of cycles of negative length.

A quick comparison of the Generic Direct Method with the Successive Approximation approach reveals why most introductory general purpose OR/MS and CS textbooks do not discuss the latter in the context of the shortest path problem, let alone in the context of a general DP framework.

5. A DP perspective on DA

It is crystal clear why DA is commonly regarded as a *greedy algorithm:* it deploys a greedy rule to determine what city should be processed next: $j = \arg\min\{F(i) : i \in U\}$. But this is only a part of the story, as after all a very important feature of DA is the manner in which the $F(\cdot)$ values are updated. To appreciate the implications of the full story we must therefore also examine the way DA updates F. So let us compare how DA and the Push procedure update F when city j is processed:

Dijkstra	$F(i) = \min\{F(i), F(j) + D(j,i)\}, i \in A(j) \cap U$
Push	$F(i) = \min\{F(i), F(j) + D(j,i)\}, i \in A(j)$

The discrepancy is that Push updates the $F(\cdot)$ values for all the immediate successors of j, whereas DA updates only the $F(\cdot)$ values of the immediate successors of j that have not been processed yet.

The explanation for this discrepancy is obvious: the Push procedure described above is not designed specifically for cases where the inter-city distances are non-negative, nor for any special rule for selecting the next city to be processed. In particular, Push – unlike DA – allows cities to be processed more than once should the need arise.

Note that if inter-city distances are non-negative and in Push we use the greedy selection rule deployed by DA, then — as in DA – each city will be processed at most once and consequently we replace the rule for updating F in Push by that used in DA.

What all this adds up to is that from a DP perspective DA is a specialization of Push for cases where:

– Select_From (Q)= $\arg\min\{F(j) : j \in Q\}$
– The inter-city distances are non-negative.

Remarks

1. The relationship between set Q used in Push and set U used in DA is $Q = \{i \in U : F(i) < \infty\}$. Since it is not necessary to actually process cities in $U \backslash Q$, we prefer to formulate DA as shown in Table 5.

Table 5. Dijkstra's algorithm, version 3

Initialize:	$F(1) = 0$; $F(j) = \infty$, $j \in C \backslash \{1\}$; $Q = \{1\} \bigcup A(1)$
Iterate:	While $(\|Q\| > 1)$ Do:
	$j = \arg\min\{F(i) : i \in Q\}$
	$Q = Q \backslash \{j\}$
	for $i \in A(j) \bigcap Q$ Do:
	$G = \min\{F(i), F(j) + D(j,i)\}$
	If$(G < F(i))$ Do:
	$F(i) = G$
	$Q = Q \bigcup \{i\}$
	End Do
	End Do
	End Do

2. The relationship between Push and DA raises the following interesting Chicken/Egg question: is Push a generalization of DA or DA is a specialization of Push? Be it as it may, from a purely historical point of view DA is definitely a specialized successive approximation method for solving the DP functional equation of the shortest path problem. This Push-type specialization exploits the fact that if the inter-city distances are not negative then the choice of the next city to be processed stipulated by $j = \arg\min F(i) : i \in U$ not only makes sense conceptually, but also performs well in practice, especially if suitable data structures are use to speed up this selection procedure.
3. To keep the discussion focussed and brief we have deliberately avoided relating DA to *Reaching* (Denardo, 2003) and *label setting* (Bertsekas, 1991; Evans and Minieka, 1992). For the record we note that in the case of cyclic networks with nonnegative distances, Reaching (Denardo, 2003, 20) is equivalent to DA and DA is clearly a label setting method (Bertsekas, 1991, 68; Evans and Minieka, 1992, 83).
4. We add in passing that Reaching and label setting methods – and for that matter label correcting methods – are all successive approximation methods. Indeed, what they all have in common is that an initial approximation of f is repeatedly updated (improved) until a fixed point of the DP functional equation is found.

5. Cormen et al. (1990) use the term *relaxation* to describe successive approximation. Their "Relax" operation (Cormen et al., 1990, 520) is equivalent to Push.
6. The same algorithm can be both of a greedy type and of a DP type. Indeed, DP provides a very natural paradigm for the formulation and analysis of greedy algorithms (see Bird and De Moor, 1997; Lew, 2005).
7. In view of this, we can describe DA as a greedy, Push type, DP successive approximation algorithm.

In short, from a DP perspective, DA can be regarded as a clever application of Push-type successive approximation methods for the solution of the functional equation associated with shortest path problems whose distances are non-negative.

6. The missing-link riddle

How come that the obvious relationship between DA and DP is persistently ignored/overlooked in OR/MS and CS courseware?

It seems that the answer to this intriguing question lies in the manner in which Dijkstra formulated the algorithm in his original 1959 paper. So, it is essential to say a few words now on this paper as far as the exposition of the algorithm is concerned.

The first thing to observe is that the paper is very short (2.5 pages long) and has a pure plain-text format. It contains no mathematical expressions or symbols, no flow charts, no figures. It does not even contain an illustrative example!

Given these features, it is not surprising that the proposed algorithm had not been identified immediately for what it is. The true nature of the algorithm was hidden in the plain-text format of the exposition.

What is more, the term "dynamic programming" is not mentioned at all in the paper and there is not even one single reference to any explicit DP publication, let alone to Bellman's (1957) first book on DP, or any one of his many published DP articles at that time (the first DP paper was published in 1952, Bellman, 1952).

Yet, not only is the connection to DP definitely there, albeit somewhat hidden, it is substantial and fundamental. To see this it is sufficient to read the first paragraph in the paper dealing with the shortest path problem. It reads as follows (Dijkstra, 1959, 270):

> **Problem 2.** Find the path of minimum total length between two given nodes P and Q.
>
> We use the fact that, if R is a node on the minimal path from P to Q, knowledge of the latter implies the knowledge of the minimal path from P to R.

This is a somewhat cryptic formulation of a typical, common *textbook* interpretation of Bellman's Principle of Optimality in the context of shortest path problems. For example, in Winston (2004, 967-8) we find a much more informative formulation:

> In the context of Example 3, the Principle of Optimality reduces to the following: Suppose the shortest path (call it R) from city 1 to 10 is known to pass through city i. Then the portion of R that goes from city 1 to city 10 must be a shortest path from city i to city 10.

There are other similar interpretations of the principle in the context of the shortest path problem (e.g. Denardo, 2003, 14-15).

Given all of this, the following question is unescapable: what is the explanation for the DP-free nature of Dijkstra's exposition of his famous algorithm?

Needless to say, we can only speculate on this issue here. But it seems that the following is a reasonable explanation.

As clearly indicated by its title, Dijkstra's paper addresses two problems. The second is the shortest path problem. The first is as follows:

> **Problem 1.** Construct the tree of minimum total length between the n nodes. (A tree is a graph with one and only one path between every two nodes)

That is, the first problem addressed by Dijkstra in his famous paper is the famous *minimum spanning tree problem.*

The similarities between the two problems and the respective algorithms proposed by Dijkstra for their solution and especially Dijkstra's explicit reference to Kruskal's Algorithm (Kruskal, 1956) strongly suggest that Dijkstra regarded his proposed algorithm for the shortest path problem as a modification of Kruskal's Algorithm for the minimum spanning tree problem. This connection may also explain why the referee(s) of his paper did not insist on the inclusion of an explicit reference to DP in the discussion on the proposed algorithm for the shortest path problem.

7. A bit of history

The literature makes is abundantly clear that the procedure commonly known today as *DA* was discovered in the late 1950s, apparently independently, by a number of analysts. There are strong indications that the algorithm was known in certain circles before the publication of Dijkstra's famous paper. It is therefore somewhat surprising that this fact is not manifested today in the "official" title of the algorithm.

Briefly: Dijkstra (1959) submitted his short paper for publication in *Numerische Mathematik* in June 1959. In November 1959, Pollack and Wiebenson (1960) submitted a paper entitled *Solutions of the shortest-route problem - a review* to the journal *Operations Research.* This review briefly discusses and

compares seven methods for solving the shortest path problem. Dijkstra's (1959) paper is not mentioned in this review.

However, the review presents a highly efficient method, attributed – as a dateless private communication – to Minty. The procedure to find the shortest path from city A to city B is described as follows (Pollack and Wieberson, 1960, 225):

> 1. Label the city A with the distance 'zero,' Go to 2.
>
> 2. Look at all one-way streets with their 'tails' labelled and their 'heads' unlabelled. For each such street, form the sum of the label and the length. Select a street making this sum a minimum and place a check-mark beside it. Label the 'head' city of this street with the sum. Return to the beginning of 2.
>
> The process may be terminated in various ways: (a) when the city B is labelled, (b) when all cities have been labelled, or (c) when the minimum of the sums is 'infinity.'

This is none other than DA!

According to Dreyfus (1969, 397), the origin of this algorithm probably dates back to Ford and Fulkerson's work on more general (network flow) problems (e.g. Ford and Fulkerson, 1958, 1962).

It is interesting to note that in his paper Dijkstra (1959, 271) refers only to two methods for solving the shortest path problem, namely Ford (1956) and Berge (1958). As we already indicated, it is extremely odd that there is no reference to DP in general, and the *Principle of Optimality* in particular, in this paper.

Another interesting historical fact is that in November 1959 George Dantzig (1960) submitted a paper entitled "On the shortest route through a network" for publication in *Management Science.* This short (3.5 pages) paper proposes an algorithm for the shortest path problem in cases where distances from each node can be easily sorted in increasing order and it is easy to ignore certain arcs dynamically. It has no reference to Dijkstra's paper, but obviously does have an explicit reference to Bellman's work (Bellman, 1958) – they both worked at the RAND corporation at that time.

The only reference these two papers (Dijkstra, 1959; Dantzig, 1960) have in common is Ford's (1956) RAND report on *Network Flow Theory.*

8. Conclusions

Dijkstra's Algorithm is covered extensively in the OR/MS and CS literature and is a popular topic in introductory CS textbooks. For historical reasons the algorithm is regarded by many as a "computer science method". But as we have shown in this discussion, this algorithm has very strong OR/MS roots. It is a pity, therefore, that the algorithm is not more widely used in introductory

OR/MS courseware and that its DP connexion is not spelled out in introductory CS courseware.

Indeed, one of the purposes of this paper is to point out that this situation can be altered for the benefit of both dynamic programming and Dijkstra's Algorithm.

OR/MS and CS courseware developers are encouraged to re-consider the way they treat this algorithm and how they relate it to other OR/MS methods and techniques. In particular, this algorithm can be used to illustrate the deployment of successive approximation methods by dynamic programming.

References

Ahuja, R.K., Magnanti, T.L. and Orlin, J.B. (1993) *Network Flow Theory, Algorithms, and Applications*. Prentice-Hall, Englewood-Cliffs, NJ.

Bellman, R. (1952) On the theory of dynamic programming. *Proceedings of the National Academy of Sciences*, **38**, 716–719.

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

Bellman, R. (1958) On a routing problem. *Quarterly of Applied Mathematics* **16** (1), 87–90.

Berge, C. (1958) *Theorie des Graphes et ses Applications*. Dunod, Paris.

Bertsekas, D. (1991) *Linear Network Optimization*. The MIT Press, Cambridge, MA.

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

Brassard, G. and Bratley, P. (1988) *Algorithmics*. Prentice-Hall, Englewood Cliffs, NJ.

Cormen, T.H., Leiserson, C.E. and Rivest, R.L. (1990) *Introduction to Algorithms*. The MIT Press, Cambride, MA.

Daellenbach, H.G., George, J.A. and McNickle, D.C. (1983) *Introduction to Operations Research Techniques*, 2nd Edition. Allyn and Bacon. Boston.

Dantzig, G.B. (1960) On the shortest path route through a network. *Management Science* **6**, 187–190.

Denardo, E.V. and Fox, B.L. (1979) Shortest-route methods: 1. Reaching, pruning, and buckets. *Operations Research* **27**, 161–186.

Denardo, E.V. (2003) *Dynamic Programming*. Dover, Mineola, NY.

Dijkstra, E.W. (1959) A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* **1**, 269–271.

Dreyfus, S. (1969) An appraisal of some shortest-path algorithms. *Operations Research* **17**, 395–412.

Evans, J.R. and Minieka, E. (1992) *Optimization Algorithms for Networks and Graphs*. Marcel Dekker, NY.

Ford, L.R. (1956) *Network Flow Theory*. RAND paper P-923.

Ford, L.R. and Fulkerson, D.R. (1958) Constructing maximal dynamic flows from static flows. *Operations Research* **6**, 419–433.

Ford, L.R. and Fulkerson, D.R. (1962) *Flows in Networks*. Princeton University Press, Princeton, NJ.

Gallo, G. and Pallottino, S. (1988) Shortest path algorithms. *Annals of Operations Research* **13**, 3–79.

Gass, S.I. and Harris, C.M. (1996) *Encyclopedia of Operations Research and Management Science*. Kluwer, Boston, Mass.

Hillier, F.S. and Lieberman, G.J. (1990) *Introduction to Operations Research*, 5th Edition. Holden-Day, Oakland, CA.

Kruskal, J.B., Jr. (1956) On the shortest spanning tree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society* **7** (1), 48–50.

Lawler, E.L. (1976) *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Whinston, NY.

Lew, A. (2005) Canonical greedy algorithms and dynamic programming. *Journal of Control and Cybernetics*, this issue.

Markland, R.E. and Sweigart, J.R. (1987) *Quantitative Methods: Applications to Managerial Decision Making*. John Wiley, New York.

Pollack M. and Wiebenson, W. (1960) Solution of the shortest-route problem – a review. *Operations Research* **8**, 224–230.

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

Winston, W.L. (2004) *Operations Research Applications and Algorithms*, Fourth Edition. Brooks/Cole, Belmont, CA.

Appendix

The discussions and tips contained in this appendix were compiled specifically for the benefit of lecturers and students.

Tip # 1
It is instructive to let students experiment with simple but "non-trivial" DP functional equations before they learn the details of DA. The following is a simple example of such a case.

Exercise. Solve the DP functional equation associated with the shortest path problem depicted in Fig. 2. For your convenience the system of equations is provided.

$$f(1) = 0$$
$$f(2) = 4 + f(4)$$
$$f(3) = \min\{2 + f(2), 1 + f(1)\}$$
$$f(4) = 3 + f(3)$$

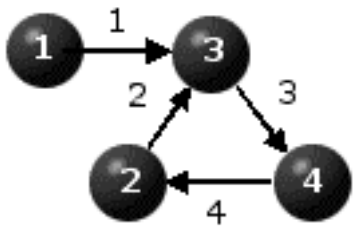


Figure 2

Note that the purpose of the exercise is not merely to find a solution to this system. After all, it is obvious (by inspection) that $f(1) = 0$; $f(2) = 8$; $f(3) = 1$;$f(4) = 4$ is a solution. Rather, the purpose is to highlight the fact that the system itself does not constitute a procedure for determining the values of $\{f(j) : j = 1, 2, 3, 4\}$ associated with this system.

Tip # 2

It is important to advise students that successive approximation methods deployed by DP are adaptations of the generic successive approximation approach commonly used in the solution of equations. This is particularly important in cases where students are already familiar with other applications of this approach. This author's favourite examples are

$$q(x) = x^2 + \beta q(x) \ , \ x \in \Re \ , \ 0 < \beta < 1$$

and

$$q(x) = x^2 + q(\beta x) \ , \ x \in \Re \ , \ 0 < \beta < 1$$

where in each case q is an (unknown) real valued function on $\Re$.

The second case can be used as a framework for drawing the students' attention to the fact some functional equations possess more than one solution and that in such cases it is necessary to decide which one is applicable to the problem in question.

Preparation of DP/DA courseware

In the context of our discussion, two related issues deserve attention in the preparation of courseware dealing with DA:

- The DP roots of this algorithm.
- The relationship between this algorithm and other types of algorithms used to solve DP functional equations.

Needless to say, the manner in which these issues should be handled would depend very much on the specific objectives of the courseware, the students' technical background, mode of delivery and many other aspects of the total teaching/learning system. For this reason we do not intend to propose a general purpose recipe for dealing with this generic dilemma.

Instead, we refer the interested reader to Lawler's (1976, Chapter 3) book as an *example* of how this can be done in a student-friendly manner. In the introduction to the chapter entitled *Shortest Paths* Lawler states the following (Lawler, 1976, 60):

> The dominant ideas in the solution of these shortest-path problems are those of DP. Thus, in Section 3 we invoke the "Principle of Optimality" to formulate a set of equations which must be satisfied by shortest path lengths. We then proceed to solve these equations by methods that are, for the most part, standard DP techniques.
>
> This situation is hardly surprising. It is not inaccurate to claim that, in the deterministic and combinatorial realm, dynamic programming is primarily concerned with the computation of shortest paths in networks with one type of special structure or another. What distinguishes the networks dealt with in this chapter is that they have no distinguishing structure.

DA makes its appearance in Section 5 of that chapter, entitled *Networks with Positive Arcs: Dijkstra's Method*.

In this context it is also appropriate to say something about what should be regarded as an "undesirable" treatment of the dilemma. In particular, there seems to be no justification whatsoever to completely separate the discussion on DA from the discussion on DP in the same courseware (e.g. a textbook). Unfortunately, this happens too often: in many OR/MS and CS textbooks there are no cross-references whatsoever between the DP section/chapter to the DA section/chapter. This, we argue should be regarded as "highly undesirable".

Note that we do not argue here that in the case of coursewares dealing with both DP and DA the discussion on the latter must be incorporated in the discussion on the former. Rather, what we argue is that in such cases the relationship between the two must be addressed, or at least alluded to.

Regarding coursewares that deal with DA but do not deal with DP, students should be advised on the DP connection and be supplied with appropriate references.

Since DA is definitely a legitimate DP "topic" it is an appropriate item for inclusion in DP coursewares. However, the question whether to incorporate DA in such coursewares must be dealt with on a case by case basis, and the answer should reflect, among other things, the specific orientation of the courseware.

Remark

It is interesting to note that the "missing DP connexion" syndrome is not confined to DA. Indeed, it definitely afflicts other OR/MS topics. For example, it is quite common that discussions on the Critical Path Method (CPM) and Decision Trees in introductory OR/MS textbooks (e.g. Winston, 2004) do not mention the fact that the quantitave models at the heart of these methods are in fact DP models.