Jogging With APL Along The Shortest Path

Moshe Sniedovich and Suzanne Findlay
Department of Mathematics
The University of Melbourne
Parkville, VIC 3052 Australia
E-mail: moshe@mundoe.maths.mu.oz.au
sue@mundoe.maths.mu.oz.au

Abstract

In this paper we examine the classical shortest path problem and illustrate the modelling issues involved in formulating APL codes for a number of generalizations thereof, including **multiobjective** problems. We also comment on APL's ability to cope with algorithms of this type.

1. Introduction

The shortest path problem is one of the most fundamental problems in **operations research** (OR). In its basic form, which we shall examine first, it involves a **network** consisting of **nodes** and **directed arcs** such that each arc has a given length defined as a numeric **scalar**. The objective is to determine the shortest path connecting a given pair of nodes, where the length of a path is equal to the **sum** of the arc lengths on the path.

As an example, consider the network depicted in Figure 1 and assume that we are interested in the shortest path from node 1 to node 7. By inspection, we discover that the shortest path is (1,3,6,7) and that its length is equal to 10. We classify such a problem as an **additive, single objective**, shortest path problem: additive because the length of a path is determined by adding the lengths of the arcs on the path; single objective because there is a single optimality criterion: a path is optimal if and only if its length is the smallest among all other feasible paths connecting the given pair of nodes.

In this paper we present APL codes for solving other types of shortest path problems. As an example of the types of problems under discussion here, consider the network depicted in Figure 2. Observe that the length of an arc is a **pair** of numbers (c,r). Interpret any such pair as follows: c represents the **cost** of the arc and r its **reliability**, where reliability refers to the probability that the arc will not fail.

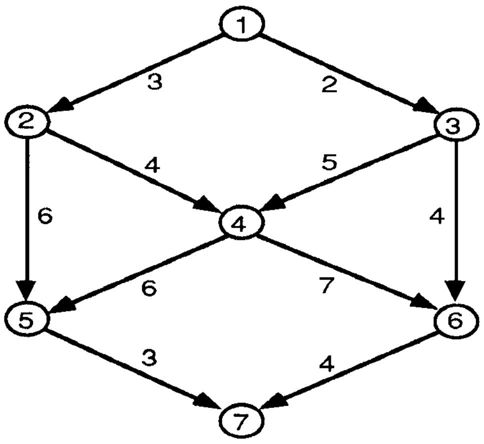

Figure 1

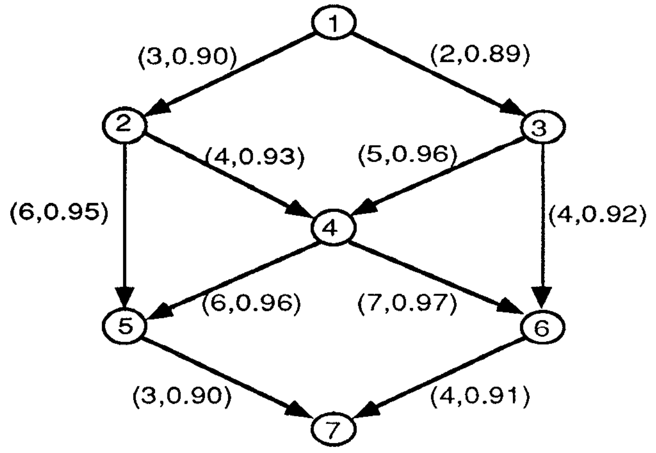

Figure 2

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission
©1992 ACM 0-89791-478-3 92/0007-0221 $1.50

So in this case we are concerned with **two** objectives: we attempt to **minimize the total cost** of a path from node 1 to node 7, while at the same time we also attempt to **maximize the overall reliability** of the path, observing that whereas the total cost of a path is equal to the **sum** of the arc lengths on the path, the overall reliability of a path is equal to the **product** of the reliabilities of the arcs on the path. Of course, in general, there is no guarantee that there exists a path that optimizes both objectives simultaneously. For example, in Figure 2, the path (1,3,6,7) minimizes the cost and the path (1,2,5,7) maximizes the reliability. There is no single path that optimizes both. What then is an optimal path?

There are a number of questions with regard to the treatment of problems of this nature. We shall focus on the following three:

> What **optimality criterion** should be used in this case?
>
> What methods can be used to **find** the best ("shortest") path?
>
> How well does **APL** cope with the tasks posed by algorithms of this type?

These are the three main topics of discussion in this paper. We begin with an outline of the structure of the additive single objective problem and the derivation of the functional equation governing the solution strategies for this problem. This done, we generalize the model and examine a number of other shortest path models. First, a model where the objective function is not additive, and then two models where there are two or more objectives. In each case we briefly discuss APL implementations of these models on the Macintosh using MicroAPL APL68000 Level II.

2. The Standard Problem

Shortest path problems consist of two interrelated constructs, namely those pertaining to the structure of the **network** underlying the problem and those pertaining to the structure of the **objective function** used to assess the relative desirability of feasible paths in the network. As far as the structure of the network is concerned, there are two basic constructs namely **nodes** and directed **arcs**.

So, let N denote the number of nodes in the network, and for simplicity assume that the nodes are labeled n=1,2,...,N with node 1 being the **origin** and node N being the **destination**. Thus, the task is to find the shortest path from node 1 to node N.

As far as the structure of the objective function is concerned, we shall begin with the assumption that the length of a path is equal to the **sum** of the arc lengths on the path.

The idea guiding the solution strategies for problems of this type is as follows: Instead of asking the question:

What is the shortest distance from node 1 to node N?

we examine the more general case:

What is the shortest distance from node 1 to some **arbitrary node** n?

In other words, we regard the problem of interest as an instance of a more general problem. So define

f(n):=the shortest distance from node 1 to node n.

And here is the crucial stage in the formulation of the solution strategy: Consider an arbitrary node that is not linked directly to the origin, call this node n. To reach this node from the origin, we must visit first one of its immediate predecessor, call it x. The question is then: what is the best x?

Since our objective is to find the shortest path, the best choice for x is the one that minimizes the sum of two quantities, namely the sum of

(1) the length of arc (x,n);

and

(2) the distance from node 1 to node x.

And so, if we let d(i,j) denote the length of arc (i,j), then clearly the best choice for x is one that minimizes the expression d(x,n)+f(x) over all x∈P(n), where P(n) denotes the set of all immediate predecessors of node n.

What emerges then is that the rule for choosing the best immediate predecessor is as follows: given node n, choose an immediate predecessor of n that minimizes d(x,n)+f(x) over P(n). In other words, the following result must be true:

Theorem

$$f(n) = \min_{x \in P(n)} \{d(x,n) + f(x)\}, \quad n=2,3,4,\ldots,N \qquad (1)$$

This is the famous **functional equation** of **dynamic programming** of Bellman [1]. Observe that for node n=1 we have f(1)=0 by definition.

The objective is then to solve this functional equation so as to determine the value of f(N) as well as to recover an optimal (shortest) path from node 1 to node N. To explain how an optimal path can be recovered, assume that we have already computed the values of f(n) for n=1,2,3,...,N and let P*(n) be the set specified as follows:

$$P^*(n) := \{y \in P(n) : d(y,n)+f(y)=f(n)\} \qquad (2)$$

namely, it is the set comprising all the immediate predecessors of node n that solve the functional equation for f(n). The recovery procedure can be formally described as follows:

```
                    Recovery Procedure:
Step 1. Initialization.
        Set x=n=N.
Step 2. Stopping Rule.
        If n=1, stop.
Step 3. Iteration.
        Set x=i,x where  i is any element of P*(n).  Set
        n=i and go to Step 2.
```

So the question boils down to this: how do we solve the functional equation?

It turns out that the question is not as simple as it may appear to be. There are two potential complications: one caused by **cycles** and one by **negative arc lengths.** We shall not elaborate on these difficulties here. The interested reader is referred to Syslo et al [4] for a detailed discussion of these difficulties and their implications.

Rather, we shall assume that the network that we are dealing with is **acyclic** in which case the **simplest** - but not necessarily the most efficient - way to solve the functional equation is to evaluate its right-hand side for n=2,3,...,N-1 - **in this order** - recalling that f(1)=0. However, to use this strategy we must ensure that the nodes are properly labeled, namely that n>x, for all x∈P(n), recalling that P(n) denotes the set of all the immediate predecessors of node n. Obviously, if the network is acyclic, it is not difficult to label the nodes properly. Therefore, in this discussion we assume that the nodes are properly labeled.

In short, we assume that the objective function is additive, the network is acyclic and the nodes are properly labeled. We refer to such a problem as a **standard shortest path problem.** As indicated above, practically all the methods developed for solving the standard shortest path problem derive, either directly or indirectly, from the dynamic programming functional equation given in (1).

Obviously, writing an APL code for this problem is not a difficult task, especially if efficiency is not a major factor. One can approach this task in different ways, depending among other things on one's personal style. The code presented below was design primarily for the purpose of exposition and as a prelude for the discussion of more complicated shortest path problems.

We assume that the network is defined by two enclosed vectors, call them p and d, each of length N, recalling that N denotes the number of nodes in the network. The first describes the architecture of the network in the following way: p[n] is a vector consisting of all the immediate successors of node n. For example, in the case of the network described by Figure 1 we can set

```
p←,¨(⍳0)(1)(1)(2 3)(2 4)(3 4)(5 6)
```

The ,¨ is used to ensure that the elements of p are all vectors, including those that for expediency were specified as scalars.

The second vector, namely d, specifies the length of the arcs. That is, for each node n, d[n] is a vector of the same shape as p[n] whose elements specify the lengths of the arcs connecting the elements of p[n] with n. Thus, in the case of Figure 2 we set

```
d←,¨(⍳0)(3)(2)(4 5)(6 6)(4 7)(3 4)
```

Next, we define a function to do exactly what the functional equation tells us we should do, except that we do it with vector notation:

```
OPT: f[⍵]←⌊/ENU ⍵
ENU: f[↑p[⍵]]+↑d[⍵]
```

That is, for each node n the function ENU generates the values of d(x,n)+f(x) for all the feasible values of x (immediate predecessors of n). The function OPT computes the smallest element of the vector computed by ENU and store the result in f[n]. By implication it is obvious that these two functions regard f as a global vector. Obviously, we shall need a function to apply OPT for every n in 1↓⍳N. But before we introduce this function, we need functions to implement the recovery procedure. So here they are:

```
OPATH: OPATH (OPTX X),⍵ : 1=X← ↑⍵ : ⍵
OPTX: ((ENU ⍵)⍳f[⍵])⌷↑p[⍵]
```

That is, the expression X←OPATH N yields the shortest path, X, connecting node 1 to node N. Observe that OPTX n yields an optimal immediate predecessor of n. In short, if we put all the pieces together, we obtain the following APL solution to the standard shortest path problem:

```
[0] R←STANDARD INPUT;p;d;f;N;⎕IO
[1] ⎕IO←1 ⋄ (p d)←INPUT ⋄ N←↑⍴p
[2] f←N⍴0
[3] R←OPT¨1↓⍳N
[4] R←f[N],OPATH N

OPT: f[⍵]←⌊/ENU ⍵
ENU: f[↑p[⍵]]+↑d[⍵]
OPATH: OPATH (OPTX X),⍵ : 1=X← ↑⍵ : ⍵
OPTX: ((ENU ⍵)⍳f[⍵])⌷↑p[⍵]
```

In the case of the network depicted in Figure 2, we obtain the following:

```
      STANDARD p d
10 1 3 6 7
```

All in all, APL does a very good job here in translating the abstract ideas underlying the algorithm into a concrete formulation.

3. Non-Additive Problems

There are many interesting problems where the length of a path is not computed as the sum of its arcs' lengths. The major ones are associated with the dyadic functions × ⌊ ⌈ . In the case of × the dynamic programming functional equation can handle only problems where the lengths of the arcs are nonnegative: if some of the lengths are negative, it is necessary to change the formulation of the model (For details see Sniedovich [3]). Although this is possible, we shall not treat this extension here. Therefore, throughout this discussion it is assumed that in the case of multiplicative objective functions the lengths of all the arcs are non-negative.

The following APL code is a simple generalization of the code presented above for the standard problem. Note that the user can specify the composition function, as well as an optimality criterion, which can be either ⌊ or ⌈ , the latter is used to solve "**longest**" path problems.

```
[0] R←(com GSP opt)INPUT;p;d;f;N;⎕IO
[1] ⎕IO← ⋄ (p d)←INPUT ⋄ N←⍴p
[2] f←N⍴(com ID)⍳0
[3] R←OPT¨1↓⍳N
[4] R←f[N],OPATH N

[0] R←(F ID)E
[1] R←(2 3 5 6=F/2 3)/(⌊/E),(⌈/E),0,1

OPT: f[⍵]←opt ENU ⍵
ENU:f[↑p[⍵]] com ↑d[⍵]
OPATH: OPATH (OPTX X),⍵ : 1=X← ↑⍵ : ⍵
OPTX: ((ENU ⍵)⍳f[⍵])⌷↑p[⍵]
```

A number of observations are in order with regard to this code. Firstly, the vector ID computes the **identity element** of the composition function com which is needed in line [2] of GSP to initialize the value of f[1]. Secondly, the functions OPTX and OPATH remained in tact. Thirdly, note that GSP is a dyadic **operator**. Fourthly, there are four valid left arguments for GSP, namely + × ⌊ and ⌈ ; and two valid right argument, namely ⌊ and ⌈ . Last but not least, the derived function + GSP ⌊ is equivalent to the function STANDARD.

And here is the code in action in the context of the network depicted in Figure 1.

```
      (+ GSP ⌊) INPUT
10 1 3 6 7
      (× GSP ⌊) INPUT
32 1 3 6 7
      (⌊ GSP ⌊) INPUT
2 1 3 4 5 7
      (⌈ GSP ⌊) INPUT
4 1 3 6 7
      (+ GSP ⌈) INPUT
18 1 2 4 6 7
      (× GSP ⌈) INPUT
336 1 2 4 6 7
      (⌊ GSP ⌈) INPUT
3 1 2 5 7
      (⌈ GSP ⌈) INPUT
7 1 3 4 6 7
```

To reiterate, the right argument of GSP determines whether the shortest (⌊) or longest (⌈) path is to be recovered; its left argument determines how the length of a path is computed from the lengths of its arcs (additive, multiplicative, etc). Thus, for instance, the derived function × GSP ⌈ will recover the longest multiplicative path, whereas ⌊ GSP ⌈ will recover the path whose shortest arc is the longest possible.

Back to the question of how well APL cope with algorithms of this type, it must be admitted that APL is really doing a splendid job here. The concept of **user-defined operators** is very natural in this context. There is, however, one small difficulty here: observe what kind of gymnastics one has to do in order to compute the identity element of a primitive function passed as an argument to a user defined operator. **APL designers should provide a more direct way to accomplish this very important task.** Although it is understood why there is no universal convention for opt/⍳0 in the case where opt is a user defined function, it is difficult to accept the fact that one cannot execute such a statement when opt is an argument of a user defined operator whose "value" is equal to a primitive function.

We now move to a much more difficult class of shortest path problems, namely problems where there are two or more objective functions.

4. Multiobjective Problems

Suppose that the lengths of the arcs are vectors of shape K such that K≥2 . In the case of Figure 2 we have K=2, where the first component of the length stipulates the cost of the arc and the second component stipulates the reliability of the arc. In view of this, the model must include additional constructs. Firstly, we must specify a composition function for each component of the length of an arc. In case of Figure 2, the first component is additive, the second is multiplicative. Secondly, for each component we have to specify an optimality criterion, namely we have to specify whether we wish to maximize or minimize the corresponding objective function. In the case of Figure 2, the first component is minimized, the second is maximized. Thirdly, we must specify some preference relation stipulating how one determines which one of a pair of vectors of shape K is better. For example, in the case of Figure 2 the length of path (1,3,6,7) is equal to `(10,0.745108)` and the length path (1,2,5,7) is equal to `(12,0.7695)`. And so, which length should we prefer? There is no obvious answer to this question because the first is better as far as the **cost** component is concerned, whereas the second is better as far as **reliability** is concerned.

But before we address these important questions we must agree on a simpler matter, namely we must decide how we expand the vector `d` to accommodate the new axis required by the fact that now the length of an arc is a vector rather than a scalar.

For reasons to become clear shortly, it is convenient to define `d` in the following way. As before we let `d` be an enclosed vector on length `N`. For each node `n` we store in `d[n]` an enclosed vector of length `K`, whose components correspond to the `K` components of the arcs. The length of each such vector is equal to the number of immediate predecessors that node `n` has. In other word, `ρ↑d[n]` is equal to `K` for all `n`, and for any given `n`, all the components of `↑d[n]` are of length `ρ↑p[n]`.

And so, in the case of Figure 2 we can specify `d` as follows:

```
   d←(ι0)((,3)(,0.90))((,2)(,0.89))((4 5)
     (0.93 0.96))((6 6)(0.95 0.96))((4 7)
           (0.92 0.97))((3 4)(0.90 0.91))
```

observing that each row corresponds to a node.

A moment of reflection on the tools of thought needed for the analysis of problems of the type we address here immediately reveals the following: what is needed here is the concept of **arrays of functions**. Since this is neither the proper place nor the proper time to discuss this concept in detail, we shall restrict the discussion only to the ad hoc codes developed to simulate this concept for the purpose of solving the shortest path problem. The interested reader is referred to APL Quote vol 14(4), 1984 for a number of papers on this subject.

The operator `ECOM` defined below is designed to facilitate the analysis and solution of multiobjective shortest path problems:

```
[0] R←A(FF ECOM)B
[1] R←(,⊂,¨FF),A B
[2] R←⍎¨⍕¨∊¨⊂[1]⊃¯1⌽R
```

Its argument is a character vector of names or symbols representing dyadic functions. This argument is treated as an array of functions, namely the derived vector of functions is applied (dyadically) to its arguments, item by item. For example,

```
      2 5 3 ('+×*' ECOM) 3 4 2
5 20 8
      A←(1 2)(4 5)
      B←(10 20)(30 40)
      DISPLAY A ('+×' ECOM) B

┌→─────────────────┐
│ ┌→────┐ ┌→──────┐ │
│ │11 22│ │120 200│ │
│ └~────┘ └~──────┘ │
└∊─────────────────┘
```

Now that we can easily manipulate arrays of functions, we can relax and think about the most important thing in multiobjective decision making, namely:

Given a collection of vectors of length k, how do you determine which one is best?

Mathematically speaking, what we need is a preference relation on the K-dimensional Euclidean space. And in the realm of decision making, there are two major alternatives namely, the **Lexicographic** order and the **Pareto** order. To explain how they work, assume that we prefer each of the components of the vectors to be as small as possible, namely we wish to minimize all the components. The lexicographic order will select the vector whose **first component** is the smallest. If there is a tie, then it is broken by considering the second component. The process continues until there is no tie, or until all the components have been used to break a tie. In APL, the Lexicographic ranking of vectors is given directly by the functions **grade up** and **grade down**. Thus, the solution is straight forward. The function `LEXICO` and its companions defined below solves the shortest path problem induced by the lexicographic preference order.

```
[0] R←LEXICO INPUT;F;N;VC;VO;p;d;J;⎕IO
[1] INITIALIZE
[2] R←FEQUATION
[3] R←F[N],RECOVER N

[0] INITIALIZE;I
[1] ⎕IO←1 ⋄ (VC VO p d)←INPUT
[2] (VC VO)←UNSPACE¨VC VO
[3] I←⊂('⌊⌈+×'ιVC) ⋄ J←¯1*'⌈⌊'ιVO
[4] F←(N←ρp)ρ⊂I⌷(⌊/ι0),(⌈/ι0),0,1

FEQUATION: OPT¨1↓ιN
RECOVER: RECOVER(OPTX X),ω : 1=X←↑ω : ω
OPT: F[ω]←⊂J×↑(↑⍋⊃A)⌷A←⊂[1]⊃J×ENU ω
ENU: (↑d[ω])(VC ECOM)(⊂[1]⊃F[↑p[ω]])
OPTX: ↑((ENU ω)ιF[ω])⌷↑p[ω]
UNSPACE: (' '≠ω)/ω
```

As we can see, except for a number of inconveniences caused by the absence of arrays of functions, the APL code cope with the task pretty well. And here is its solution to the reliability problem depicted in Figure 2.

```
      DISPLAY LEXICO '+×' '⌊⌈' p d
∘→──────────────────────∘
│ ∘→──────────∘          │
│ │10 0.745108│ 1 3 6 7  │
│ ∘~──────────∘          │
∘∊──────────────────────∘
```

Obviously, if we regard reliability, rather than cost, the more important objective, we may be interested in the solution generated by

```
      DISPLAY LEXICO '×+' '⌈⌊' p (⌽¨d)
∘→────────────────────∘
│ ∘→────────∘          │
│ │0.7695 12│ 1 2 5 7  │
│ ∘~────────∘          │
∘∊────────────────────∘
```

Next, let us examine the situation where the preference order is of the **Pareto** type. The idea underlying this preference is outlined by the following :

> **Definition.**
> Let **x** and **y** be any vectors in the k-dimentional Euclidean space. We say that **x** is **dominated** by **y** if and only if
>
> (1) $x_i \le y_i$ for all i, $1 \le i \le k$; and
> (2) $x_i < y_i$ for at least one i, $1 \le i \le k$.

Thus, given a finite collection of vectors, we do not attempt to find the "optimal" element of this collection, but rather seek to identify **all** the nondominated elements of the collection. The point is that the collection may have several nondominated elements.

This means that the Pareto approach does not actually provide a solution to the shortest path problem because it may fall short of selecting a single path. It merely eliminate paths that in a sense are inferior to other feasible paths. Some other criteria must be used to pinpoint the shortest path.

Observe that in the case of the Pareto preference order we formally define

f(n):= **Set** of nondominated paths from node 1 to node n, n=1,2,...,N.

So now APL is confronted with a functional equation involving sets of vectors rather then single vectors. This, however, does not cause any major difficulty because **enclosed arrays** can handle the situation reasonably well. The complete code is as follows:

```
[0] R←PARETO INPUT;N;⎕IO;VC;VO;p;d;J;F
[1] INITIALIZE
[2] R←FEQUATION
[3] R←(↑F[N])(N RECOVER¨↑F[N])

[0] INITIALIZE;I
[1] ⎕IO←1 ⋄ (VC VO p d)←INPUT
[2] (VC VO)←UNSPACE¨VC VO
[3] I←⊂('⌊⌈+×'⍳VC) ⋄ J←¯1*'⌈⌊'⍳VO
[4] F←(N←⍴p)⍴⊂,⊂I⌷(⌊/⍳0),(⌈/⍳0),0,1

[0] R← A (FF ECOM) B
[1] R←((↑⍴B)⍴¨⊂¨A FF),,⊂B
[2] R←↑(⍎¨)¨(⍕¨)¨⊂[1]¨⊃¨⊂[1]⊃R

[0] R← U OPTX n ;I;Z
[1] I←↑,/(⊂,U←⊂↑,/U)∊¨Z←MENU n
[2] R←↑(I←↑I/⍳⍴Z)⌷↑p[n←↑n]
[3] Z←,⊂(I⌷¨↑d[n])(VC ECOM)(↑F[R])
[4] R←R,↑(Z⍳U)⌷↑F[R]

FEQUATION: OPT¨1↓⍳N
OPT: F[⍵]←J×(↑⍋⊃A)⌷A←J×ENU ⍵
MENU: (⊂[1]⊃↑d[⍵])ENU¨F[↑p[⍵]]
ENU: (⊂⍺))(VC ECOM),¨⊂¨⍵
UNIQUE: ((⍵⍳⍵)=⍳⍴⍵)/⍵
UNSPACE: (' '≠⍵)/⍵
RECOVER: ((↑Z),⍺)RECOVER 1↓Z←⍵ OPTX X :
                                   1=X←↑⍺ : ⍺
OPT: F[⍵]←⊂(⊂J)×(~∨/(M∧.≥⍉M)∧(M∨.>⍉M←⊃A
                ))/A←UNIQUE (⊂J)×↑,/MENU ⍵
```

To determine the nondominated paths associated with the reliability problem represented by Figure 2, we apply the code to the same input vector used previously by `LEXICO`. The result is reshaped to prevent overflow.

```
   DISPLAY 2 1⍴PARETO '+×' '⌊⌈' p d
∘→─────────────────────────────────∘
│ ∘→────────────────────────────∘ │
│ │ ∘→─────────∘ ∘→───────────∘ │ │
│ │ │12 0.7695│ │10 0.745108│ │ │
│ │ ∘~─────────∘ ∘~───────────∘ │ │
│ ∘∊────────────────────────────∘ │
│ ∘→──────────────────────∘       │
│ │ ∘→───────∘ ∘→───────∘ │       │
│ │ │1 2 5 7│ │1 3 6 7│ │       │
│ │ ∘~───────∘ ∘~───────∘ │       │
│ ∘∊──────────────────────∘       │
∘∊─────────────────────────────────∘
```

In other words, there are two nondominated paths `(1 2 5 7)` and `(1 3 6 7)`. the first costs `12` units and its reliability is equal to `0.7695` and the second costs `10` units and its reliability is equal to `0.745108`.

Observe that the same result is obtained if the elements of the input vector are reversed, namely

```
 DISPLAY 2 1⍴PARETO 'x+' '⌈⌊' p (⌽¨d)
∘→-----------------------------∘
| ∘→-------------------------∘ |
| | ∘→---------∘ ∘→--------∘ | |
| | |10 0.745108| |12 0.7695| | |
| | ∘~---------∘ ∘~--------∘ | |
| ∘∊-------------------------∘ |
| ∘→--------------------∘      |
| | ∘→------∘ ∘→------∘ |      |
| | |1 3 6 7| |1 2 5 7| |      |
| | ∘~------∘ ∘~------∘ |      |
| ∘∊--------------------∘      |
∘∊-----------------------------∘
```

Generally speaking, APL is doing a reasonable job here in coping with the operations required by the dynamic programming algorithm, except that life could have been a bit easier had we had arrays of functions at our disposal.

5. Conclusions

In this investigation we examined APL in a context where it should perform extremely well, namely in a situation where it is used as a notation for the formulation of numeric algorithms. It therefore should not come as a surprise that overall its performance proved excellent, yet not perfect. While the present discussion is confined to the analysis of shortest path problems, it has been our experience that similar conclusions apply in the case of other operations research problems, Sniedovich [2].

Acknowledgement. The first author's work in this area was partially supported by an ARC grant SG6890197.

References

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

2. M. Sniedovich, **The APL Phenomenon: An Operational Research Perspective**, European Journal of Operational Research, 38(2), pp. 141-145, (1989).

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

4. M.M. Syslo, N. Deo, and J. Kowalik, **Discrete Optimization Algorithms with Pascal Programs**, Prentice-Hall, Englewood Cliffs, NJ (1983).