Boolean Data in Dyalog APL

Lars Wentzel 2022-01-04

Unfamiliar with APL? You might still be able to follow me in the syntax below. You just have to see how I use arrays and especially vectors like 4 5 6. 4. This is a vector (list) of three numeric elements (values). An example of a three-character vector is `'cde'`.

Basics

APL in general does not have typed data. Boolean data is not different. It is just like any other numeric data. It is just found to be Boolean if the values are only 0 or 1. Boolean data usually comes from comparisons like:

```
      'abcde'∈'be'      (belongs to)
0 1 0 0 1
Or
      5>3 4 5 6 7       (larger than)
1 1 0 0 0
```

It can of course be used in Boolean operations

```
       1 0 1∧1 1 0      (and)
1 0 0
       1 0 1∨1 1 0      (or)
1 1 1
```

If you have a vector (string or list) of Boolean numbers:

```
bool←0 1 0 0 1 1 1 0 0
```

It can be used as any other numeric data i.e., it can be summarized.

```
      +/bool
4
```

But then it can also be used for selection.

```
       string←'abcdefghi'
       bool/string
befg
```

Or for any array – matrix or higher

```
      names
Adam
Bertil
Cesar
David
Erik
Filip
Gustav
Helge
Ivar
       bool/[1]names    (first dimension i.e., rows)
Bertil
Erik
Filip
Gustav
```

The index numbers of the values with 1 can easily be found

Length of bool

```
      ⍴bool
9
```

Indexes of length

```
      ⍳⍴bool
1 2 3 4 5 6 7 8 9
      bool/⍳⍴bool
2 5 6 7
```

All these basic functions make a very big difference to all programming. When you start learning how to use this it will have a high impact on the code written.

But now we can go a bit further. But first an understanding of the implementation in Dyalog APL. Booleans are stored with one bit for each value. This means that you get 8 Boolean values per byte. This makes storing and handling extremely space efficient. The next implementation issue is the optimization of the functions using Booleans making the execution very, very fast.

Here comes examples of different and more advanced use of Booleans.

Products vs capacities and remaining capacity per capacity id and week

You have a number of *products* that are produced (50111).
Then you have a number of *capacity constraints* (597). These are production, logistic and, market capacities. A production capacity is typically a maximum number to produce something in a machine or a line. It can also be the working time. A logistic capacity is the supply of parts. A market capacity is some kind of delivery restriction, either a market quota or transportation capacity.

The problem is to find the first production slot of each product and how many can be produced.

Capacities are expressed as remaining capacity per week since part of the total capacity is booked by planned production of already ordered products.
Below is a short example showing remaining capacity per week of five capacities.

Capacity	2021w46	2021w47	2021w48	2021w49	2021w50	2021w51	2021w52
1	0	100	150	150	150	150	150
2	0	150	200	200	300	300	300
3	5	0	30	40	90	90	90
4	0	50	0	60	30	0	45
5	0	30	40	30	0	101	99

Then you have the cross matrix where products are mapped to capacities

	Capacity				
Product	1	2	3	4	5
A	1	1	0	0	0
B	0	1	0	1	0
C	1	0	0	0	0
D	0	0	1	0	1
E	0	1	0	0	1
F	0	0	1	1	0
G	0	1	1	1	0

E.g., product A will use capacities 1 and 2. Product G will use 2,3 and 4.

Product G will then have the following weekly availability

Capacity	2021w46	2021w47	2021w48	2021w49	2021w50	2021w51	2021w52
2	0	150	200	200	300	300	300
3	5	0	30	40	90	90	90
4	0	50	0	60	30	0	45
Min	0	0	0	40	30	0	45

To find the available capacity for this product you need to take the lowest value above
This means that that a first production slot is in week 49 where 40 products G can be produced.

What you want is this information for all products. In general, this is a three-dimensional problem: Products, capacities and weeks. This is also the way I tried to solve it. But it turned out to become too large a problem. Instead, I found this solution.

Variables:

prod

A
B
C
D
E
F
G

capac

1
2
3
4
5

week

2021w46
2021w47
2021w48
2021w49
2021w50
2021w51
2021w52

avail (available capacity per week)

0	100	150	150	105	150	150
0	150	200	200	300	300	300
5	0	30	40	90	90	90
0	50	0	60	30	0	45
0	30	40	30	0	101	99

cross (products vs. capacities)

1	1	0	0	0
0	1	0	1	0
1	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	0	1	1	0
0	1	1	1	0

And the calculation is

```
min←⊃[2]⌊⌿¨(⊂[2]cross)⌿¨⊂avail
```

With the result

```
      prod min
┌→──────────────────────────────────────┐
│ ┌→┐  ┌→──────────────────────────────┐ │
│ ↓A│  ↓0 100 150 150 105 150 150      │ │
│ │B│  │0  50   0  60  30   0  45      │ │
│ │C│  │0 100 150 150 105 150 150      │ │
│ │D│  │0   0  30  30   0  90  90      │ │
│ │E│  │0  30  40  30   0 101  99      │ │
│ │F│  │0   0   0  40  30   0  45      │ │
│ │G│  │0   0   0  40  30   0  45      │ │
│ └─┘  └~──────────────────────────────┘ │
└~──────────────────────────────────────┘
```

Or formatted

Product	2021w46	2021w47	2021w48	2021w49	2021w50	2021w51	2021w52
A	0	100	150	150	105	150	150
B	0	50	0	60	30	0	45
C	0	100	150	150	105	150	150
D	0	0	30	30	0	90	90
E	0	30	40	30	0	101	99
F	0	0	0	40	30	0	45
G	0	0	0	40	30	0	45

For a non-APLer this is of course difficult to understand. This is an alternative to writing a loop.
I will take it step by step.

```
⊂[2]cross
```

This will make a vector (list) of the rows in the cross matrix i.e., a vector of vectors (list of lists).

```
┌→──────────────────────────────────────────────────────────────────────────┐
│ ┌→────────┐ ┌→────────┐ ┌→────────┐ ┌→────────┐ ┌→────────┐ ┌→────────┐ ┌→────────┐ │
│ │1 1 0 0 0│ │0 1 0 1 0│ │1 0 0 0 0│ │0 0 1 0 1│ │0 1 0 0 1│ │0 0 1 1 0│ │0 1 1 1 0│ │
│ └~────────┘ └~────────┘ └~────────┘ └~────────┘ └~────────┘ └~────────┘ └~────────┘ │
└~──────────────────────────────────────────────────────────────────────────┘
```

Then I use ¨ which means each. So, each of these vectors will select rows from avail e.g.

```
      1 1 0 0 0⌿avail
0 100 150 150 105 150 150
0 150 200 200 300 300 300
      0 1 0 1 0⌿avail
0 150 200 200 300 300 300
0  50   0  60  30   0  45
```

Then comes the next operation. That is to find the minimum per column of each of these matrixes.

```
⌊⌿¨
```

```
0 100 150 150 105 150 150
0  50   0  60  30   0  45
```

The result is a vector of vectors

```
┌→──────────────────────────────────────────────┐
│ ┌→──────────────────────┐ ┌→──────────────┐ … │
│ │0 100 150 150 105 150 150│ │0 50 0 60 30 0 45│ │
│ └~──────────────────────┘ └~──────────────┘   │
└~──────────────────────────────────────────────┘
```

```
⊃[2]
```
Then I make this a matrix where each vector becomes a row. And then I have my result

```
┌→──────────────────────────────┐
↓0 100 150 150 105 150 150      │
 0  50   0  60  30   0  45      │
└…──────────────────────────────┘
```

A nice and compact expression. Isn't it?

In my real example I had 50111 products and 597 capacities.
The calculation takes **46 ms** in my computer with version 18.0 of Dyalog APL. This is down with 50% since version 17. This an exceptional performance and demonstrates the optimization that can be made in an array interpreter. This is a practical solution although my Boolean matrix is not so dens, only about 4%.

The size of the cross matrix is `29,916,267` elements, occupying the space of `3,739,556` bytes. This means that you can store this data using very little space.

Now you can also get lots of other interesting information easily out of this.
Number of capacities hit per product

```
      +/cross
2 2 1 2 2 2 3
```

Number of products hit by each capacity

```
      +/[1]cross
2 4 3 3 2
```

List of products for capacity 2

```
    cross[;2]⌿prod
A
B
E
G
```

BOM problem. Orders vs. BOM-rules

The example is that you have a complicated configurable product with an almost unique Bill of Material for each order. You have around 2000 parts in each product. The total number of parts is 5000. This means that you have a high degree of re-use i.e., the configured products have a lot of common parts. Actually, one part can exist in more than one place in a product and there can be more than one part in each location. You have 6000 part-rules for these 5000 parts.

So, imagine you have 100,000 orders of these products and 6000 part-rules. Then you have a cross matrix of 100,000 x 6,000. This is a fairly dense Boolean matrix with a size of 75 MB. This makes this very space-efficient to store and handle. If you handle it as a combinations of orders and BOM-rules it would require 100,000 x 2,000 records of 8 bytes each. This will be around 1.6 GB.

BOM rules

Id	part	place	number
1	A531	abc	1
2	A531	ukt	2
3	B810	ukt	1
4	A830	avb	3
…			

Orders

Id
X001
X002
Y003
Y856
Z125

And then you have the cross matrix of Orders vs. BOM rules

```
       ord_bom
1  0  0  1
1  1  0  1
0  1  1  0
0  1  1  1
1  0  0  1
```

Or formatted

	Rule			
Order	1	2	3	4
X001	1	0	0	1
X002	1	1	0	1
Y003	0	1	1	0
Y856	0	1	1	1
Z125	1	0	0	1

Without the Boolean matrix with would become:

```
 Ord   Rule
 X001  1
 X001  4
 X002  1
 X002  2
 X002  4
 Y003  2
 Y003  3
 Y856  2
 Y856  3
 Y856  4
 Z125  1
 Z125  4
```

From the Boolean cross matrix data you can also easily pick the BOM list for some orders:

```
  (⊂[2]ord_bom)⌿¨bom[2]
 A531  A531  A531  A531  A531
 B830  A531  B810  B810  B830
       B830        B830
```

And the number for each

```
     (⊂[2]ord_bom)⌿¨bom[4]
 1  1  2  2  1
 3  2  1  1  3
    3     3
```