APL2 and Artificial Intelligence

Parallel and Search Papers

AI Programming in APL2: General Search Techniques

Dr. James A. Brown, APL Chief Architect, and Edward V. Eusebi, Knowledge Based Systems, IBM Santa Teresa Lab J88/B25, 555 Bailey Ave., P.O. Box 50020, San Jose, Calif. 95150 USA

ABSTRACT

This paper discusses search strategies and shows how they can be implemented in *APL2* without regard to the particular problem being addressed. The three strategies demonstrated are Depth First Search, Breadth First Search, and Best First Search. They will be exercised by applying them to an 8 puzzle. The implementations take advantage of *APL2* functional style and the ability to pass functions as parameters to programs.

Additional information about the use of *APL2* for AI applications can be found in (Br1).

INTRODUCTION TO SEARCH

Complex logical problems, real life problems, and games all present you with similar situations -- you are in some starting position, and you want to reach some goal. In a logic problem, you might be given a set of logic statements and the goal might be to generate a particular new logic expression. In real life, you might have a set of facts about your financial situation and the goal might be a decision to buy or not buy a car. In a game the goal might be to win against an opponent. In each case, from the starting position, there may be many first steps that you can take. From each of these, there may be many additional steps you can take. Eventually, you expect one of the steps to take you to the desired goal. There could be many paths that lead to the goal, there could be one unique path that leads to the goal, or there might be no path at all in which case the goal cannot be reached.

Given a set of next steps that you could take in searching for a goal, the search strategies define organized ways to tell you which step to try next.

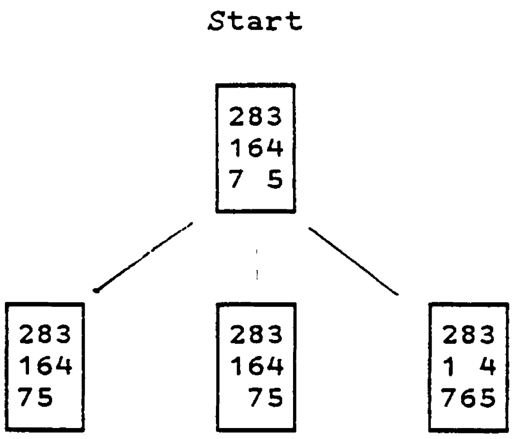
SEARCH AND THE 8 PUZZLE

The 8 puzzle is normally packaged in a flat tray partitioned into a nine squares. Eight of the nine positions have tiles numbered 1 through 8. One of the squares is, therefore, empty. The challenge is to take the puzzle with some given starting arrangement of tiles, and rearrange them into some required new arrangement by sliding tiles into the empty spot. Here is an example starting arrangement and an example goal:

```
    Start        Goal

    283          123
    164          8 4
    7 5          765
```

In starting to solve this puzzle, there are exactly three possible first moves:



From the first two of these positions, you can produce two more arrangements (two of which give the initial arrangement). From the third position, you can produce four more arrangements.

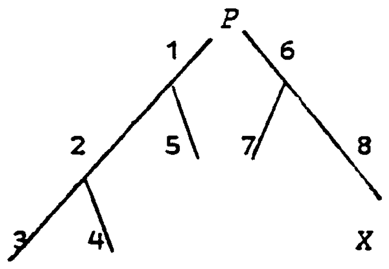
Programs that apply search strategies to the solution of this problem are presented later.

SEARCH STRATEGIES

The search strategies determine the order in which possible next steps in a search are taken. Only three simple search methods are discussed here: Depth First Search, Breadth First Search, and Best First Search.

** DEPTH FIRST SEARCH

In a depth first search, if there are two paths to be tried, then every possible path arising after the first one is taken, is tried before any other path is tried. Here's a tree showing the order in which paths will be taken in trying to reach goal X from starting position P:



Because the leftmost path was taken first, every path from it is taken before the other path from P is tried. Notice that no knowledge of the problem influences the order of the search.

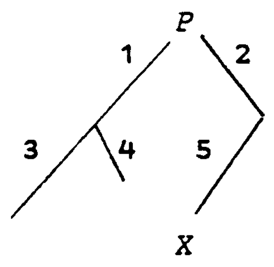
If you are trying to reach *X*, the order of paths makes a significant difference in the amount of work to be done. Also, if the path starting with 1,2,3 went on infinitely long, a depth first search would not find the path to *X* even though it existed. The 8 puzzle encounters this problem with a Depth First Search.

Suppose you wanted to get from the airport to a hotel in the middle of a city to the west of the airport. A depth First Search may choose to go north one block first. Since the hotel is not reached, the search proceeds by going another block north. Eventually, you can't go any further north so you back up and move in some other direction. You will eventually reach the hotel but only after taking a lot of unnecessary roads. Of course, if the search started to the west, the search would succeed much faster. Since no knowledge of the problem is used, this could happen only by accident.

You can treat the paths waiting to be tried in a Depth First Search as a push down stack where the last paths put on the stack are the first ones taken out (FIFO).

** BREADTH FIRST SEARCH

If there are two paths to be tried, then the second is tried after the first but before any paths following from the first. Here's a tree showing the order in which paths will be taken in trying to reach goal X from starting position P:



In this case, search stops because *X* was found. A breadth first search is guaranteed to find a path if one exists and if there is a finite number of possible paths at each step.

Suppose you wanted to get from the airport to a hotel in the middle of a city to the west of the airport. A Breadth First Search is at least bounded -- you'll spread out in a radius about the starting point and eventually find the hotel. You may again take a lot of unnecessary roads, but you will reach your destination.

You can treat the paths waiting to be tried in a Breadth First Search as a push down stack where the first paths put on the stack are the first ones taken out (FIFO).

** BEST FIRST SEARCH

Much of the challenge in search programs is to find better search strategies that use some knowledge of the situation to make smarter choices of what to try next. By computing a "figure of merit" with each possible path, you can choose an apparent best next choice and significantly reduce the amount of work done. Even a very bad figure of merit can lead to a vast improvement in efficiency.

One such search strategy is called a Best First Search (in particular, this one is called A* (Po1)). It measures the cost of a path by the amount of actual work done so far plus the estimate of how much more work there is to do. If the estimate is any good at all, the search will try many fewer paths. As the estimate gets more accurate, the number of paths tried shrinks.

You can think of the set of paths tried so far as a push down stack as before accept, this time, the order in which paths are selected depends on knowledge about the problem.

Here is a simplified outline of how Best First Search works:

1. Begin with the start position as a zero length path

2. Choose the path with smallest cost

    - If the stack is empty, no path exists

    - If more than one path with smallest cost, choose one

    - If chosen path reaches goal, stop

3. Delete the path from the stack and add all new paths composed of one more step.

4. Delete from the stack any path that reaches a spot reached cheaper by another path.

5. Go to 2

In the map example, a Best First Search would move out from the airport in all possible directions but then choose the next step to take based on the position that is closer to the hotel. You might still run into a dead end and have to back up but, in general, you will visit very few unnecessary roads.

IMPLEMENTING THE SEARCH STRATEGIES

It is possible to write special search programs for special problems. Here, general search programs are written that do not have any knowledge of any particular problem built in.

The search programs must be called with the following information:

- A starting position

- A program that can compute the next position and say if it is a goal

- For *BESTFIRST* - a function that estimates cost of reaching the goal.

For the 8 puzzle, the starting position can be a 3 by 3 character matrix. Note that boxes are used to make the structure of the data apparent in these examples. Real *APL*2 output would not include these boxes unless a function like *DISPLAY* were specifically coded:

```
        START←3 3ρ'2831647 5'
        START
  ┌───┐
  │283│
  │164│
  │7 5│
  └───┘
```

The program that computes the set of next positions is called *MOVE*8. It takes a position as its argument and returns a list of new positions and identifies which if any are goals. It always computes the new positions by trying the four actions:

- move a tile to the left
- move a tile to the right
- move a tile up
- move a tile down

```
        MOVE8 START
  ┌─────────────────────┐     ┌─────┐
  │ ┌───┐  ┌───┐  ┌───┐ │     │0 0 0│
  │ │283│  │283│  │283│ │     └─────┘
  │ │164│  │164│  │1 4│ │
  │ │75 │  │ 75│  │765│ │
  │ └───┘  └───┘  └───┘ │
  └─────────────────────┘
```

This result is a two item vector where the first item is three new positions and the second item is three zeros meaning none ofthe new positions is the goal.

The estimator function takes a position as argument and returns a number that indicates its distance from the goal using some arbitrary measurement. The function *EST*1 counts the number of tiles out of place:

```
        EST1 START
  5
```

The function *EST*2 measures the rectangular distance of each tile from its final position:

```
        EST1 START
  6
```

This second measure is a more accurate measure of the amount of work to be done. The *BESTFIRST* search should be more efficient given the better estimator.

The result of the search function is a path that leads from the starting position to the goal if one exists.

Here is an example of each search program applied to the 8 puzzle:

```
        (MOVE8 DEPTHFIRST) START
```

Notice that the program *DEPTHFIRST* is given the function *MOVE*8 and the initial data *START* as parameters. A program that can be given functions as parameters is called a defined operator in *APL*2.

DEPTHFIRST first moves a tile to the left. On the next iteration, it will move a tile to the right giving the initial arrangement again. It will never get out of this loop.

```
        (MOVE8 BREADTHFIRST) START
   283    283    2 3     23    123    123
   164    1 4    184    184     84    8 4
   7 5    765    765    765    765    765
```

The program had to iterate through 278 paths in order to find this one.

The value of an estimate is shown by using a function *EST*1 that computes the number of tiles out of place:

```
        (MOVE8 BESTFIRST EST1) START
   283    283    2 3     23    123    123
   164    1 4    184    184     84    8 4
   7 5    765    765    765    765    765
```

The program only iterated through 7 paths.

A better estimate is the distance of a tile from it's correct position. Since moves are only horizontal an vertical, this distance is an integer. Here's *BESTFIRST* using this estimate of cost:

```
      (MOVE8 BESTFIRST EST2) START

   283    283    2 3     23    123    123
   164    1 4    184    184     84    8 4
   7 5    765    765    765    765    765
```

This is, of course, the same answer but only required 6 paths to be examined thus proving the value of a good estimator even in a trivial situation.

Each of the search program internally uses a stack to keep track of paths that have been generated but not yet examined. Here is the essential logic of the search programs:

1. Select a path to examine
   - If Depth or Breadth first - select top path on stack
   - If Best first - select lowest cost path
2. Quit if path reaches the goal
3. Compute next positions
4. If Best First - compute cost estimate to goal
5. Add new positions to the stack
   - If Depth First - new positions to top of stack
   - If Breadth First - new positions to bottom of stack
   - If Best First - new positions to top of stack (arbitrary)
6. If Best first - delete any paths reached by other paths at lower cost
7. Go to 1

The actual programs are in the Appendix.

The stack maintained internally is a nested matrix with 2 columns (for Depth and Breadth First) or 4 columns (for Best First). Here is what the stack looks like after two iterations of a Breadth First Search:

```
┌───────────────┐
│ ┌───┐ ┌───┐   │            0
│ │283│ │283│   │
│ │164│ │164│   │
│ │ 75│ │7 5│   │
│ └───┘ └───┘   │
└───────────────┘
┌───────────────┐
│ ┌───┐ ┌───┐   │            0
│ │283│ │283│   │
│ │1 4│ │164│   │
│ │765│ │7 5│   │
│ └───┘ └───┘   │
└───────────────┘
┌─────────────────────┐
│ ┌───┐ ┌───┐ ┌───┐   │      0
│ │283│ │283│ │283│   │
│ │164│ │164│ │164│   │
│ │7 5│ │75 │ │7 5│   │
│ └───┘ └───┘ └───┘   │
└─────────────────────┘
┌─────────────────────┐
│ ┌───┐ ┌───┐ ┌───┐   │      0
│ │283│ │283│ │283│   │
│ │16 │ │164│ │164│   │
│ │754│ │75 │ │7 5│   │
│ └───┘ └───┘ └───┘   │
└─────────────────────┘
```

This is a four by 2 matrix. Column 1 contains the paths generated but not yet examined, and column two is 0 meaning that the path does not reach a goal. The matrix for a Best First Search would have two additional columns giving the costs of producing the path and the estimate of the distance to the goal.

Refer to the Appendix for the details of the programs.

CONCLUSION

This paper has introduced some relatively simple search techniques. They are useful as demonstrations of the power of *APL*2. A look at the programs shows that they are short and match closely the outline of the algorithms. They make significant use of nested arrays and defined operators. All three programs can be applied to any search problem because no problem specific information is imbedded in the logic of the programs.

ACKNOWLEDGEMENT

The original *BESTFIRST* program and its application to the 8 puzzle was written by Chuck Haspel, IBM Systems Research Institute.

REFERENCES

(Br1) Brown, J., Eusebi, E., Groner, L., Cook, J., "Algorithms for Artificial Intelligence in APL2", IBM Santa Teresa Technical Report TR 03.281

(Po1) Pohl, Ira, "Heuristic Search Viewed as Path Finding in a Graph", Artificial Intelligence, Vol. , No. 3, 1970.

(Ri1) Rich, Elaine,"Artificial Intelligence", McGraw-Hill Books, New York, New York.

APPENDIX: APL2 PROGRAMS

DEPTH FIRST SEARCH

The Depth First search program follows the general search outline.

```
      ∇ Z←(MOVE DEPTHFIRST)START;B;T;MAT;NEWP;GLS
  [1]   MAT←1 2⍴(,⊂START)0
  [2]   Z←⍳0
  [3]  LOOP:→(0=↑⍴MAT)/0
  [4]   →(2⊃B←MAT[1;])/DONE
  [5]   (NEWP GLS)←MOVE 1 1⊃B
  [6]   T←((⊂¨NEWP),¨B[1]),[1.5]GLS
  [7]   MAT←T,[1]1↓[1]MAT
  [8]   →LOOP
  [9]  DONE:Z←⌽↑MAT[1;1]
```

Line 1 initializes the stack matrix so it has one row and two columns. The first item is a one item list containing the start position. The second item is a zero meaning that the start position is not a goal. Line 2 sets the program result variable to an empty vector. This becomes the result of the program on line 3 should the stack matrix ever become empty (meaning that no path to the goal exists). Line 4 selects the top row of the stack and exits if it is a goal. Line 5 selects the current ending position of the selected path and produces all possible next positions by the appropriate *MOVE* routine (passed as a parameter). Line 6 builds one new path for each new position reached by the move routine. Line 7 deletes row 1 and adds the new paths to the top of the stack. Thus, this program implements a depth first search because on each iteration, one row is taken off the top of the matrix and zero or more new rows are put back on the top of the matrix.

BREADTH FIRST SEARCH

Breadth First is identical to Depth First except that it puts new paths on the bottom of the stack matrix.

```
      ∇ Z←(MOVE BREADTHFIRST)START;B;T;MAT;NEWP;GLS
  [1]   MAT←1 2ρ(,⊂START)0
  [2]   Z←⍳0
  [3]  LOOP:→(0=↑ρMAT)/0
  [4]   →(2⊃B←MAT[1;])/DONE
  [5]   (NEWP GLS)←MOVE 1 1⊃B
  [6]   T←((⊂¨NEWP),¨B[1]),[1.5]GLS
  [7]    MAT←(1↓[1]MAT),[1]T
  [8]   →LOOP
  [9]  DONE:Z←⌽↑MAT[1;1]
```

Lines 6 and 7 delete the path just examined and adds the new paths discovered to the bottom of the stack so that any existing paths will be examined before the newly generated ones.

BEST FIRST SEARCH

The Best First Search program is similar to the other two search programs. A four column stack is kept. The path selected next is the one whose actual cost plus the estimated cost is the least. The new paths are put on the top but except for the case where more than one path has the minimum cost, the logic of the program is unaffected.

```
      ∇ Z←(MOVE BESTFIRST EST)START;B;T;M;MAT;IX;NEWP;GLS
  [1]    MAT←1 4⍴(,⊂START)0(,0)(EST START)
  [2]    Z←⍳0
  [3]   LOOP:→(0=↑⍴MAT)/0
  [4]    B←MAT[IX←IX⍳⌊/IX←MAT[;4]+↑¨MAT[;3];]
  [5]    →B[2]/DONE
 ·[6]    (NEWP GLS)←MOVE 1 1⊃B
  [7]    T←((⊂¨NEWP),¨B[1]),GLS,((1+3 1⊃B),¨B[3]),[1.5]
  EST¨NEWP
  [8]    MAT←T,[1]MAT[(⍳↑⍴MAT)~IX;]
  [9]    (T U)←MAT[;1] MAT[;3]
  [10]   MAT←(∧/¨∧/¨(U∘.≤¨,/U)∨~T∘.=__¨,/T)⌿MAT
  [11]   →LOOP
  [12]  DONE:Z←⌽↑MAT[IX;1]
```

Line 1 defines a one row matrix as the initial stack. The first two columns are defined as before. Column 3 is a list giving the actual cost paid to reach each of the positions of the path in column 1. This program adds 1 to the cost for each more (on line 7) but a fancier program would have the *MOVE* program return a value for the cost. Column 4 is the estimated cost for reaching the goal. Line 4 uses the most recent actual cost and the estimated cost to the goal to choose the most likely path to extend next. This is, in general, not at the top of the stack so this program is neither depth first nor breadth first. Line 7 build s the new positions into a four column matrix and line 8 adds them to the top of the matrix. Line 9 makes sophisticated use of nested arrays to see if any final path is reached on some other path at cheaper cost. You can learn a lot of *APL*2 by figuring out these lines.

MOVE FUNCTION FOR THE 8 PUZZLE

The move function for the 8 puzzle attempts to move the blank space in each of the four possible directions. Any move that would not end up on the board are discarded. The function returns the vector of new positions and a 1 if the new position is the goal. Notice that the definition of the goal is built into the move function itself. Strictly speaking it could be a separate function. It is combined with the move function because its result is needed at the same time and because there is a limitation of at most 2 functions as parameters to an *APL*2 defined operator.

```
     ∇ Z←MOVE8 M;IB;MI;IN
[1]   ⍝ move generator for 8 puzzle
[2]    IB←(,M∊' ')/,MI←(⍳3)∘.,⍳3
[3]    IN←(IN∊MI)/IN←,(0 1)(0 ¯1)(1 0)(¯1 0)∘.+IB
[4]    Z←IN RMOVE8¨⊂M
[5]    Z←Z (Z=__¨⊂3 3⍴'1238 4765')

     ∇ Z←I RMOVE8 M
[1]   ⍝ recursive replacer, subfunction of MOVE8
[2]    (IB⊃Z)←(⊂I)⊃Z←M
[3]    ((⊂I)⊃Z)←' '
```

ESTIMATOR FUNCTIONS FOR THE 8 PUZZLE

The first estimator just counts the number of tiles not in position. The second counts the number of horizontal and vertical moves that each tile would have to make if there was nothing in the way.

```
     ∇ Z←EST1 P
[1]   ⍝ number out of position estimate for 8 puzzle
[2]    Z←+/,P≠3 3⍴'1238 4765'

     ∇ Z←EST2 P;I
[1]   ⍝ Manhattan distance estimate for 8 puzzle
[2]    Z←+/|∊I-(I←,(⍳3)∘.,⍳3)[('1238 4765')⍳,P]
```

Parallel Solutions to Logic Problems

James A. Brown

IBM Santa Teresa Lab J88/E42
555 Bailey Ave.
P.O. Box 50020
San Jose, Calif. 95150 USA

Manuel Alfonseca

IBM Madrid Scientific Center
Paseo de la Castellana, 4
28043 Madrid, SPAIN

ABSTRACT

Logic problems are traditionally solved using recursive notations. These include LISP which was invented for the solution of logic and AI related problems and more recently PROLOG. *APL* provides a different way to look at logic problems.

This paper explores array solutions to logic problems and shows how they can be solved, at least conceptually, in parallel using APL2.. Problems include puzzles from "Alice in Puzzleland" and symbolic mathematics. The solutions are contrasted with solutions to the same problems in PROLOG. Thus, it is shown that *APL2* has logic programming capabilities in addition to its well known computational abilities.

INTRODUCTION

Logic programming has been with us for more than a decade. PROLOG-like languages are presented usually, in the literature on artificial intelligence, as the preferred way to solve problems that involve some reasoning on the part of the machine, including all the set of what are generally known as logic problems (see references sm1 and sm2). PROLOG is, in fact, an elegant non-procedural language based on traditional logic.

Perhaps the most significant feature of PROLOG that makes it different to other, more traditional, programming languages is that it is non-procedural. While traditional programming languages execute in sequential order except as controlled by control structures, PROLOG instructions (see cl1) are not given nor are they evaluated in any pre-established order. The system (the "inference processor") has access to all the instructions at the same time, and selects by itself, depending on the actual values of the data, the instruction that must be executed at a given instant (the rule or axiom to be applied, in the PROLOG slang).

Unfortunately in real PROLOG programs, it is usually necessary to define the order of certain operations or to change their order depending on certain conditions. The three basic elements of structured programming (Block, If-Then-Else, and Do-While) must therefore be included somehow in the PROLOG structure. This has been done as follows:

1. A block is defined as a Horn clause where all the elements succeed always.

```
block_name <- C1 & C2 & ... & Cn.
```

   where Ci always succeeds, is equivalent to

```
begin
C1
C2
...
Cn
end
```

2. If-Then-Else is emulated by means of the "Cut" primitive:

```
condition_name <- C1 & / & C2.
condition_name <- C3.
```

   is equivalent to

```
if C1 then C2 else C3
```

3. Do-While is emulated by means of the "Repeat" and "Cut" primitives:

```
            while_name <- REPEAT & body.
            body <- C1 & / & C2 & fail.
            body.
```

is equivalent to

```
            while C1 do C2
```

In this paper we make the statement that the PROLOG-like non-procedural structure is not the only way of solving artificial intelligence problems in a "natural" way, that is to say, with a program that is legible, compact, and represents the problem immediately. We believe that *APL2* provides a different way of programming, actually a parallel way of programming, that is at least as compatible with our way of thinking as PROLOG is, but may be even more appropriate to the structure of future computing machines.

The following pages show some examples of logic programming -- an area where PROLOG is considered to be strong.

BOOLEAN LOGIC

Let *P* be a logical proposition. It has a truth value, i.e. it is either false or true.

For example,

- "All men are yellow"

  is false

- "Some men are yellow"

  is true

- "If all men are yellow then men are alive"

  is true.

- "If all men are yellow then today is Monday"

  may be true or false depending on the day of the week we are in.

In logic programming languages, you write statements that are true and then draw conclusions from them. For example, in PRO-

LOG you can say that "John likes Mary" with the following assertion:

```
        likes (John, Mary).
```

meaning that "It is true that John likes Mary".

On the other hand, *APL2* uses a different computational approach to write logic statements. The truth values "false" and "true" are represented in *APL2* as the numbers 0 and 1, respectively and you should write expressions that represent all the possible combinations of truth and falsity for the variables involved. The evaluation of the resulting expression then gives a set of "ones" and "zeros". Each "one" corresponds to a combination of the variables that make the expression true. The "zeros" represent combinations that make the expression false.

Let *P* represent a logical proposition and let *P1* represent its possible truth values. The set of all possible truth values for *P* is the two item vector:

```
      P1←0 1
```

The "negation" of *P* is defined as another proposition *P'* such that *P'* is true when *P* is false and *P'* is false when *P* is true. This means that the truth value of *P'* will be 0 when the truth value of *P* is 1, and vice versa. Therefore, the set of all possible truth values of *P'* can be represented as:

```
      ~P1
1 0
```

Given this representation, trivial expressions about *P* can be computed. For example, a tautology is always true:

```
      P1 ∨ (~P1)
1 1
```

A contradiction is never true:

```
      P1 ∧ (~P1)
0 0
```

From this point on, no distinction between the "propositional variable" (what was called *P*) and its possible truth values (what was called *P1*) will be made.

If you want to write logical expressions on two propositions, *P2*, and *Q2*, there are four possible combinations of truth values: If *Q2* is false, then *P2* may be false or true. If *Q2* is true, then *P2* may be false or true. Thus, for two variables, complete sets of values can be represented as four item vectors:

```
        P2← 0 1 0 1
        Q2← 0 0 1 1
```

Now non-trivial expressions can be written. The expression *P2∧Q2* is true only when both *P2* and *Q2* are true (the conjunction of *P2* and *Q2*):

```
        P2 ∧ Q2
  0 0 0 1
```

The expression *P2 ∨ Q2* only fails to be true when both *P2* and *Q2* are false (the disjunction of *P2* and *Q2*):

```
        P2 ∨ Q2
  0 1 1 1
```

De Morgan's law shows that the negation of a conjunction is a disjunction and vice versa. One formulation of this rule is:

```
      P2 ∨ Q2 ←→ ~ (~P2) ∧ (~Q2)
```

Computationally this is

```
        (~P2) ∧ (~Q2)
  1 0 0 0
        ~ (~P2) ∧ (~Q2)
  0 1 1 1
```

which is the "or" function.

Logical implication is defined in propositional calculus as:

"*P2* implies *Q2* is logically equivalent to the disjunction of *Q2* and the negation of *P2*." The corresponding *APL2* expression will thus be:

```
        Q2 ∨ (~P2)
  1 0 1 1
```

This result has a 1 wherever *Q2* is either greater or equal to *P2* and so implication could also be written with a single *APL2* primitive:

```
        Q2 ≥ P2
  1 0 1 1
```

Finally, the statement "*P*2 if and only if *Q*2" can be rephrased as "*P*2 implies *Q*2 and *Q*2 implies *P*2." The formula is as follows:

```
       (Q2∨(~P2)) ∧ (P2∨(~Q2))
 1 0 0 1
```

This expression is true precisely where both are false and both are true -- that is when the logic values match. Thus, this equivalence can be represented by the single *APL2* function EQUAL (=):

```
       P2 = Q2
 1 0 0 1
```

Applying this to the previous expression of De Morgan's law you see that

```
     (P2 ∨ Q2) = ~ (~P2) ∧ (~Q2)
 1 1 1 1
```

is a tautology. Thus, De Morgan's law is always true.

Expressions containing three variables have eight possible combinations of values:

```
       P3 ← 0 1 0 1 0 1 0 1
       Q3 ← 0 0 1 1 0 0 1 1
       R3 ← 0 0 0 0 1 1 1 1
```

Here is the computation of three different implications:

1. *P*3 implies *Q*3

```
       Q3 ∨ (~P3)
 1 0 1 1 1 0 1 1
```

2. *Q*3 implies *R*3

```
       R3 ∨ (~Q3)
 1 1 0 0 1 1 1 1
```

3. *P*3 implies *R*3

```
       R3 ∨ (~P3)
 1 0 1 0 1 1 1 1
```

Suppose that you claim that "*P*3 implies *Q*3" and "*P*3" are simultaneously true (Modus Ponens):

```
      (Q3 ∨ (~P3)) ∧ P3
 0 0 0 1 0 0 0 1
```

You might expect to see the representation of *Q3* from this computation (0 0 1 1 0 0 1 1). The answer differs from *Q3* where *P3* is false but *Q3* is true. Since it is claimed that *P3* is true, the boolean result is stronger than just *Q3*. It expresses the fact that both *P3* and *Q3* are true simultaneously.

Next, look at the chaining rule: If "*P3* implies *Q3*" and "*Q3* implies *R3*" then "*P3* implies *R3*". The results of the individual implications are already listed above. The computation of the chaining rule is:

```
      (Q3 ∨ (~P3)) ∧ (R3 ∨ (~Q3))
 1 0 0 0 1 0 1 1
```

Again, you might expect the representation of "*P3* implies *R3*" (1 0 1 0 1 1 1 1) but again the result produced is stronger.

Suppose that, in addition to the chaining rule, you assert that *P3* is actually true:

```
      (Q3 ∨ (~P3)) ∧ (R3 ∨ (~Q3)) ∧ P3
 0 0 0 0 0 0 0 1
```

This shows that *P3*, *Q3*, and *R3* are all simultaneously true. This is stronger than the result of "*P3*" and "*P3* implies *R3*":

```
      (R3 ∨ (~P3)) ∧ P3
 0 0 0 0 0 1 0 1
```

which makes no claim about the truth of *Q3*.

PARALLEL BOOLEAN LOGIC

This section shows how you might go about using the application of the *APL2* logical functions to solve logic problems for all solutions in parallel.

THE MARCH HARE

The following logic problem is adapted from Raymond Smullyan's book "Alice in puzzle-land" (sm2). It is solved by application of parallel boolean logic.

> "The jam had been stolen by either the March Hare, the Mad Hatter, or the Dormouse. They were arrested and each made one statement.
> They were:
>
> a.) The March Hare: I am not guilty.
>
> b.) The Mad Hatter: I am not guilty.
>
> c.) The Dormouse: At least one of the others speaks the truth.
>
> Further investigation produced the following conclusions:
>
> d.) The March Hare and the Dormouse didn't both say the truth.
>
> e.) Only one of them was guilty.
>
> Who was guilty?"

The solution to this puzzle uses the concepts of parallel logic previously developed. There are three logic variables to deal with:

- The Dormouse truth (*DT*),
- the Mad Hatter truth (*HT*),
- and the March-hare truth (*MT*).

These can be represented by the following three vectors:

```
      DT←0 1 0 1 0 1 0 1
      HT←0 0 1 1 0 0 1 1
      MT←0 0 0 0 1 1 1 1
```

These three variables represent all possible combinations of truth and falsity for the three entities.

What the Dormouse said can be expressed logically as follows: *MT*∨*HT*. The Dormouse told the truth if and only if one of the others spoke the truth. This equivalence is written in *APL2* using EQUAL. Therefore the complete expression of the Dormouse's statement is *DT*=(*MT*∨*HT*). This relationship must be true.

Condition d. must be false if both *MT* and *DT* are true. This is written: ~(*MT*∧*DT*)

Condition e. says that only one is guilty. Since two said they were not guilty one of them is correct. This is expressed as *MT*∨*HT*.

Finally, we know that d. and e. are true. since the three preceding statements are simultaneously true, the individual expressions may be "anded" together giving one expression of truth:

```
       COND←(DT=MT∨HT)∧(~DT∧MT)∧(MT∨HT)
       COND
  0 0 0 1 0 0 0 0
```

The result represents the case where what the Dormouse says is true, what the Mad Hatter says is true, but where the March Hare is lying. Thus the problem is solved. Here is the final program that solves this puzzle, which includes a print out of the results:

```
        ∇JAM
  [1] ⍝ ALICE IN PUZZLE-LAND
  [2]  DT←0 1 0 1 0 1 0 1
  [3]  HT←0 0 1 1 0 0 1 1
  [4]  MT←0 0 0 0 1 1 1 1
  [5]  COND←(DT=MT∨HT)∧(~DT∧MT)∧(MT∨HT)
  [6]  'HARE IS ',((COND/MT)/'NOT '),'GUILTY'
  [7]  'HATTER IS ',((COND/HT)/'NOT '),'GUILTY'
  [8]  'DORMOUSE IS ',((~(COND/MT)∧(COND/HT))/'NOT '),'GUILTY'
      ∇
```

Lines 6, 7 and 8 can be explained as follows: *COND* says which of the possible combinations of truth values is compatible with the data. Therefore, the expression *COND/MT* gives the truth value of the statement by the Hare (i.e. whether the Hare said the truth or lied). Since the Hare actually stated its own innocence (statement a.) line 6 is immediate. So is line 7, which applies the same discussion to the Hatter. Finally, the Dormouse is guilty if and only if both the Hare and the Mad Hatter are not guilty, that is to say, if both said the truth ((*COND/MT*)∧(*COND/HT*).)

Here is the execution of the program:

```
      JAM
   HARE IS GUILTY
   HATTER IS NOT GUILTY
   DORMOUSE IS NOT GUILTY
```

Here is the PROLOG solution of the same problem for comparison:

```
  guilty(hare)<--¬true(hare).
  guilty(hatter)<--¬true(hatter).
  guilty(dormouse)<--¬guilty(hare)&¬guilty(hatter).
  true(dormouse)<-true(hare)|true(hatter).
  ¬(true(hare)&true(dormouse)).
  true(hatter)|true(hare).
```

Invocation is:

```
    <-guilty(*).
  (i.e. who is guilty)
  SUCCESS: guilty(hare).
```

<u>ALICE</u>

The following problem, taken from (sm1) is a little different:

"When Alice entered the forest of forgetfulness,
she did not forget everything, only certain things.
She often forgot her name, and the most likely
thing for her to forget was the day of the week.
Now, the lion and the unicorn were frequent visitors
to this forest. These two are strange creatures.
The lion lies on Mondays, Tuesdays, and Wednesdays,
and tells the truth on the other days of the week.
The unicorn, on the other hand, lies on Thursdays,
Fridays, and Saturdays, but tells the truth on the
other days of the week.

One day Alice met the lion and the unicorn resting
under a tree. They made the following statements:

LION: Yesterday was one of my lying days
UNICORN: Yesterday was one of my lying days

From these statements, Alice, who was a bright girl,
was able to deduce the day of the week. What was it?"

First the data must be defined. Here the variable *DAYS* is defined as the seven days of the week and *YEST* is defined as the day before each day of the week:

```
    DAYS←'Sun' 'Mon' 'Tue' 'Wed' 'Thu' 'Fri' 'Sat'
    YEST←'Sat' 'Sun' 'Mon' 'Tue' 'Wed' 'Thu' 'Fri'
```

Next, two variables are set up that describe the days when the lion lies (*LL*) and the days when the unicorn lies (*UL*):

```
         LL ← 'Mon' 'Tue' 'Wed'
         UL ← 'Thu' 'Fri' 'Sat'
```

Now you must write expressions that are true. There are two conditions under which the lion statement is coherent. Either this is one of his truth telling days and yesterday was a lying day or this is one of his lying days and yesterday was a truth telling day. Here are the boolean expressions that compute both of these:

```
         (~DAYS∊LL)     ∧     (YEST∊LL)
         1 0 0 0 1 1 1 ∧ 0 0 1 1 1 0 0
  0 0 0 0 1 0 0
         (1 0 0 0 1 1 1 ∧ 0 0 1 1 1 0 0)/DAYS
  Thu
         (DAYS∊LL)      ∧     (~YEST∊LL)
         0 1 1 1 0 0 0 ∧ 1 1 0 0 0 1 1
  0 1 0 0 0 0 0
         (0 1 1 1 0 0 0 ∧ 1 1 0 0 0 1 1)/DAYS
  Mon
```

This says that if the lion is telling the truth it could only be Thursday and if the Lion is lying then this could only be Monday. Thus, we may define a variable representing when the lion statement is coherent (*LC*):

```
        LC ← ((~DAYS∊LL)∧(YEST∊LL)) ∨ ((DAYS∊LL)∧(~YEST∊LL))
```

The same logic is true for the unicorn:

```
        (~DAYS∊UL)     ∧     (YEST∊UL)
        1 1 1 1 0 0 0 ∧ 1 0 0 0 0 1 1
  1 0 0 0 0 0 0
        (1 1 1 1 0 0 0 ∧ 1 0 0 0 0 1 1)/DAYS
  Sun
        (DAYS∊UL)      ∧     (~YEST∊UL)
        0 0 0 0 1 1 1 ∧ 0 1 1 1 1 0 0
  0 0 0 0 1 0 0
        (0 0 0 0 1 1 1 ∧ 0 1 1 1 1 0 0)/DAYS
  Thu
```

Here's the expression for the coherence of the unicorn statement (*UC*):

```
        UC ← ((~DAYS∊UL)∧(YEST∊UL)) ∨ ((DAYS∊UL)∧(~YEST∊UL))
```

By inspection you can see that only Thursday is true in both cases. Here, then is a summary of the solution in a more compact form:

```
    YEST ← ¯1⌽DAYS←'Sun' 'Mon' 'Tue' 'Wed' 'Thu' 'Fri' 'Sat'
   (LL UL) ← ('Mon' 'Tue' 'Wed')('Thu' 'Fri' 'Sat')
        LC ← ((~DAYS∊LL)∧(YEST∊LL)) ∨ ((DAYS∊LL)∧(~YEST∊LL))
        UC ← ((~DAYS∊UL)∧(YEST∊UL)) ∨ ((DAYS∊UL)∧(~YEST∊UL))
        (LC∧UC)/DAYS
  Thu
```

This problem can therefore be solved using entirely boolean expressions in parallel written to describe precisely the problem as stated.

Sullivan and Fordyce (fo1) describe a clever scheme for implementing a production expert system in *APL* using Boolean logic.

Here is the PROLOG solution of the same problem for comparison. The logic is close to the *APL* implementation:

```
  yest(sun,sat).
  yest(mon,sun).
  yest(tue,mon).
  yest(wed,tue).
  yest(thu,wed).
  yest(fri,thu).
  yest(sat,fri).
  ll(mon).
  ll(tue).
  ll(wed).
  ul(thu).
  ul(fri).
  ul(sat).
  lc(*day) <- yest(*day,*yest) & ll(*day) & ¬ ll(*yest).
  lc(*day) <- yest(*day,*yest) & ll(*yest) & ¬ ll(*day).
  uc(*day) <- yest(*day,*yest) & ul(*day) & ¬ ul(*yest).
  uc(*day) <- yest(*day,*yest) & ul(*yest) & ¬ ul(*day).
  get(*day) <- lc(*day) & uc(*day).
  <- get(*day).
```

AGES

Here is a puzzle that can be solved by logic programming languages but where *APL2* can express a much more direct and efficient solution:

"The sum of the ages of John and Mary is 40,
and their difference is 6.

What are their ages?"

This kind of problem can be solved using the methods of logic programming but it can also be expressed as a set of linear equations as follows:

```
    40 = John + Mary
     6 = John - Mary
```

This is easily solved in *APL* as follows:

```
      COEFF ← 2 2⍴ 1 1 1 ¯1
      COEFF
1  1
1 ¯1

      40 6 ⌹ COEFF
23 17
```

This kind of problem, that requires a certain number-crunching capacity, is not amenable to an "effective" solution by a classical logic language such as PROLOG.

Here is the PROLOG solution:

```
     is_integer(1).

     is_integer(*X) <- is_integer(*Y) & sum (1,*Y,*X).

     ages(*John,*Mary) <- is-integer(*John) &
                          sum(*John,*Mary,40) &
                          diff(*John,*Mary,6).
     <- ages(*John,*Mary).
   ages(23,17).
```

The problem with the PROLOG solution is the method of solution. First, the system assigns variable John a value of 1. Then it tries to find a value of Mary that complies line 3. It can't, so it tries for John the value of 2. And so on, until it tries value 23 for John, the condition holds, and the answer is given. (But what if the answer would have been 1E6?. This program would need a million trials to reach it).

The following more direct PROLOG statement is rejected by all commercial PROLOG systems known to the writers.

```
        ages(*John,*Mary) <- sum(*John,*Mary,40)
                             & diff(*John,*Mary,6).
```

SYMBOLIC DERIVATION

Non-procedural languages, such as PROLOG, are well suited for symbol manipulation applications. *APL2* is also well suited for symbol manipulation as shown by the following discussion of symbolic differentiation.

The following set of functions computes the derivative of a symbolic expression with respect to a variable. Expressions including addition (+), subtraction (-), multiplication (×), division (÷), power (*) and natural logarithm (LOG(...)), but without parenthesis (apart from those in the call to LOG) are differentiated correctly, assuming the following function hierarchy: power, division, multiplication, subtraction and addition. Monadic negation (-) is also accepted, and applies to everything on its right.

Here are a few simple examples:

```
      'X' D 'X*3'
3×X*2
      'X' D 'LOG(X)'
X* ¯1
```

The functions that implement this derivative are listed below for completeness. A few comments on implementation follow but the details are not important. What is important is that a few pages of relatively straightforward *APL2* implements this operation.

```
⍝ DERIVATIVE
      ∇ Z←X D FX;U;V;C
  [1]    →(PLUS,MINUS,TIMES,DIV,POWER)IF '+-×÷*'∊FX←,FX
  [2]    →LOG IF 'LOG('∧.=4↑FX
  [3]    →CONST IF IS_A_CONST FX
  [4]    →ERROR IF 1≠⍴FX
  [5]    Z←⍕FX=X
  [6]    →0
  [7]   CONST:Z←'0'
  [8]    →0
  [9]   PLUS:Z←(X D '+' HEAD FX)SPLUS(X D '+' TAIL FX)
  [10]   →0
  [11]  MINUS:→NEG IF FX[1]='-'
  [12]   Z←(X D '-' HEAD FX)SMINUS(X D '-' TAIL FX)
  [13]   →0
  [14]  NEG:Z←'-',X D 1↓FX
  [15]   →0
  [16]  TIMES:V←'×' TAIL FX
  [17]   →CTIMES IF IS_A_CONST U←'×' HEAD FX
  [18]   Z←(U STIMES(X D V))SPLUS(V STIMES(X D U))
  [19]   →0
  [20]  CTIMES:Z←U STIMES X D V
  [21]   →0
  [22]  DIV:Z←X D('÷' HEAD FX)STIMES('÷' TAIL FX),'*¯1'
  [23]   →0
  [24]  POWER:→ERROR IF~IS_A_CONST C←'*' TAIL FX
  [25]   Z←C STIMES(X D U)STIMES(U←'*' HEAD FX)SPOWER⍕¯1+⍎C
  [26]   →0
  [27]  LOG:→ERROR IF ')'≠¯1↑FX
  [28]   Z←(U SPOWER ¯1)STIMES X D U←4↓¯1↓FX
  [29]   →0
  [30]  ERROR:Z←'ERROR'
      ∇
```

```
⍝ SIMPLIFIED ADDITION
      ∇ Z←A SPLUS B
  [1]    →ACONS IF IS_A_CONST A
  [2]    →BCONS IF IS_A_CONST B
  [3]    →NORED IF~A≡B
  [4]    Z←'2×',A
  [5]    →0
  [6]   ACONS:→ABCONS IF IS_A_CONST B
  [7]    →NORED IF 0≠⍎A
  [8]    Z←B
  [9]    →0
  [10]  BCONS:→NORED IF 0≠⍎B
  [11]   Z←A
  [12]   →0
  [13]  ABCONS:Z←⍕(⍎A)+⍎B
  [14]   →0
  [15]  NORED:Z←A,'+',B
      ∇

⍝ SIMPLIFIED SUBTRACTION
      ∇ Z←A SMINUS B
  [1]    →ACONS IF IS_A_CONST A
  [2]    →BCONS IF IS_A_CONST B
  [3]    →NORED IF~A≡B
  [4]    Z←'0'
  [5]    →0
  [6]   ACONS:→ABCONS IF IS_A_CONST B
  [7]    →NORED IF 0≠⍎A
  [8]    Z←'-',B
  [9]    →0
  [10]  BCONS:→NORED IF 0≠⍎B
  [11]   Z←A
  [12]   →0
  [13]  ABCONS:Z←⍕(⍎A)-⍎B
  [14]   →0
  [15]  NORED:Z←A,'-',B
      ∇
```

```
⍝ SIMPLIFIED MULTIPLICATION
      ∇ Z←A STIMES B
  [1]    →ACONS IF IS_A_CONST A
  [2]    →BCONS IF IS_A_CONST B
  [3]    →NORED IF~A≡B
  [4]    Z←A,'*2'
  [5]    →0
  [6]   ACONS:→ABCONS IF IS_A_CONST B
  [7]    →ZERO IF 0=⍎A
  [8]    →NORED IF 1≠⍎A
  [9]    Z←B
  [10]   →0
  [11]  BCONS:→ZERO IF 0=⍎B
  [12]   →NORED IF 1≠⍎B
  [13]   Z←A
  [14]   →0
  [15]  ABCONS:Z←⍕(⍎A)×⍎B
  [16]   →0
  [17]  ZERO:Z←'0'
  [18]   →0
  [19]  NORED:Z←A,'×',B
      ∇

⍝ SIMPLIFIED POWER
      ∇ Z←A SPOWER B
  [1]    →ACONS IF IS_A_CONST A
  [2]    →BCONS IF IS_A_CONST B
  [3]   NORED:Z←A,'*',B
  [4]    →0
  [5]   ACONS:→ABCONS IF IS_A_CONST B
  [6]    →ONE IF 1=⍎A
  [7]    →NORED IF 0≠⍎A
  [8]    Z←'0'
  [9]    →0
  [10]  BCONS:→ONE IF 0=⍎B
  [11]   →NORED IF 1≠⍎B
  [12]   Z←A
  [13]   →0
  [14]  ABCONS:Z←⍕(⍎A)*⍎B
  [15]   →0
  [16]  ONE:Z←'1'
      ∇
  ⍝ AUXILIARY FUNCTIONS
        ⍝ GET ALL OF X AT THE LEFT OF N
        ∇ Z←N HEAD X
    [1]   Z←(¯1+(X=N)⍳1)↑X
        ∇

        ⍝ GET ALL OF X AT THE RIGHT OF N
        ∇ Z←N TAIL X
    [1]   Z←((X=N)⍳1)↓X
        ∇

        ∇ Z←A IF B
    [1]   Z←B/A
        ∇

        ⍝ TEST WHETHER X IS A CONSTANT
        ∇ Z←IS_A_CONST X
    [1]   →0 IF 0=Z←∧/X∊'¯0123456789.'
    [2]   X←'Z←0' ⎕EA X
        ∇
```

Here are some more examples of the use of the derivitive function:

```
        'X' D '2×X*3-3×X*2×Y+X×Y*2-Y*3'
  2×3×X*2-3×Y×2×X+Y*2

        'Y' D '2×X*3-3×X*2×Y+X×Y*2-Y*3'
  -3×X*2+X×2×Y-3×Y*2

        'X' D 'X+1÷X'
  1+¯1×X*¯2

        'X' D 'X+LOG(X)÷X'
  1+LOG(X)×¯1×X*¯2+X*¯1×X*¯1
```

It can be easily ascertained that the derivatives are correct, although the final result is not simplified. The way in which the *D* function performs derivation is very legible and straightforward, and follows directly the rules for mathematical derivation. Line 1 selects the appropriate function to be differentiated, according to the indicated order of precedence. If this function is a plus sign, line 9 receives control. This line directly applies the rule that "the derivative of a sum is the sum of the derivatives of the terms of the sum. In fact, the expression used in line 9

```
    [9]  PLUS:Z←(X D '+' HEAD FX)SPLUS(X D '+' TAIL FX)
```

just tells that the derivative of the sum is the symbolic sum (performed by function *SPLUS*, that also adds a little simplification, such as eliminating zeros and adding constants) of the derivative with respect to the same variable of the head of the sum (function *HEAD* extracts from its right argument everything to the left of its left argument) plus the derivative of the tail of the sum (function *TAIL* extracts the corresponding right part of its right argument).

A simple inspection of the remainder of the *D* function will demonstrate that the other functions are derived precisely in the same way, with *APL2* lines that immediately represent the corresponding derivation rules.

This differentiation program is not presented here as a real application, and may contain errors or inconsistencies in certain cases. The only reason why it has been done was to be an example and show the way in which *APL2* can perform symbol manipulation operations with as much ease and legibility as other languages theoretically designed only or mainly for that purpose.

CONCLUSION

We believe that the above examples show that the applicability of *APL2* to Artificial Intelligence has probably been underestimated, and should be redefined. This is not to say that *APL2* should be the language of choice for every possible problem in Artificial Intelligence. No language is good for everything, and all languages are specially suitable for something. A PROLOG solution may be a much better choice for a given application. What we are stating is that *APL2* should be considered as one of the standard possibilities for the design of Artificial Intelligence applications, to be selected or rejected on the basis of actual, practical considerations.

Finally, *APL2* itself could be extended in some way to make it more useful for those problems (if any) where it is not optimally applicable at the moment.

REFERENCES

- cl1: W.F. Clocksin, C.S. Mellish, "Programming in Prolog", Springer-Verlag, 1981.

- fo1: Fordyce, K., Sullivan, G. ,"Artificial Intelligence Development Aids (AIDA)", Proceedings of Lapl.APL85, *APL* Quote QUad, Vol. 15, No. 4, 1985, pp.106-113.
- sm1: Raymond Smullyan, "What is the name of this book?"
- sm2: Raymond Smullyan, "Alice in puzzle-land"