CRL

BNR

TITLE : **Costas Arrays**

REFERENCE : 90079

DATE : 29 November 1990

AUTHOR : André Vellino

ABSTRACT :

The implementation of Costas Arrays in Refine (the language of the Kestral Institute Development System-KIDS) is compared to solutions written in Prolog (with `freeze`), Pascal and CHIP.

COSTAS ARRAYS IN REFINE

In his paper on automated software development [Smith 90], Smith discusses the automatic generation by Refine of a program to solve a constraint problem called Costas-arrays. A Costas array is a permutation of N integers such that there are no duplicates in any row of the difference table. The first row of the difference table gives the difference of the adjacent elements, the second row gives the difference of every second element and so on. For example, the following is a Costas array of dimension 5:

```
1   5   3   2   4
  -4  2   1  -2
    -2  3  -1
      -1  1
         3
```

Smith reports that the Refine "theory" that specifies Costas-arrays took about a week to formulate and that the automated deduction of the algorithm from the specification takes about 25 minutes to produce in Refine. The resulting un-optimized code that Refine produces takes about 50 minutes to find all 760 costas-arrays of size 9 using a Sun-4/160. Computer assisted optimizations produce a version of this algorithm that takes 5 minutes for the same problem and hand-coding the same algorithm in C reduces this execution time to about a 1 second.

Since the algorithm that was chosen to solve this combinatorial problem in Refine was essentially a backtracking algorithm (branch and bound), it seemed interesting to compare the performance of a Prolog solution to this problem. This natually led to the question of how this problem would be tackled using a conventional language such as Pascal. Thus in the following sections I describe of the steps undertaken in producing the Prolog, CHIP and Pascal programs and attempt to draw some conclusions from this experience.

COSTAS ARRAYS IN PROLOG

The naive attempt at solving the costas array problem was to use the familiar "Generate and Test" method. First you generate a permutation of N integers; then you create the first row of the difference table as a list of N-1 variables that have been constrained to be different (using `freeze`), then you start computing this row of the difference table. If any element in the difference table fails, the constraints then force backtracking. If the first row is successful then you move onto the next row etc. The full program, which took about 1 hour to write and test is less than a page of Prolog. It shown as [LP 1] in the Appendix.

This first version can be made more efficient if the *entire* difference table is constructed first and then constrained to have its elements be different (see [LP 2] in the Appendix). This clearly forces more failures sooner. However, both these methods suffer from the defect that the lower bound on the complexity is bounded by the permutation part of the algorithm (N!).

Much greater pruning of the search tree can be achieved by constraining the calculation of the difference table to fire whenever a new element of the permutation is added to the initial sequence, i.e. by freezing the arithmetic constraints on the elements on the diference table. This has the effect of interleaving the permutation and the testing, i.e. making it a "Test and Generate" algorithm. This was done by making simple modifications to the original program and is shown in [LP 3] in the Appendix.

The timing results for [LP 3] on a Mac-IIci compare reasonably well with the un-optimized results produced by Refine (see Table 1). If we assume that a SPARCstation is between 3 and 4 times faster than a Mac-IIci, this effictively means that the Prolog version takes about twice as long.

COSTAS ARRAYS IN PASCAL

Essentially the same algorithm was implemented in Pascal [P1] by Jérôme Chiabaut. It took about an hour to write the program and it is a factor of about 100 times faster than the equivalent Prolog program. These results are consistent with previous experiments on similar comparisons done on algorithms for the N-Queens problem.

2

An optimized version of this algorithm [P2] was re-written in about the same time as the first version and runs almost twice as fast. It is interesting to note that this program is faster than the hand-coded, 'optimized' C program derived from Refine, even though it takes no advantage known symmetries in the problem.

COSTAS ARRAYS IN CHIP

The Costas arrays problem was also solved using the constraint logic programming language CHIP [van Hentenryck, 89]. The program that describes the costas array problem took between one and two hours to develop and runs about 20 times more slowly than the C program output by Refine.

An interesting point to note is that the 'look-ahead' algorithms built into CHIP are not useful for solving the Costas array problem.

	Refine*	Refine->C*	LP2	LP 3	P1	P2	CHIP*
Costas (9)	300 s	5s	10440 s	2100 s	18s	10s	150s

* Refine and CHIP benchmarks run on a SPARCstation. All the others were run on a Mac-II-ci.

Table 1

CONCLUSIONS

It is hard to know quite what to conclude from this benchmark excercise. It is clear that the Costas array problem is small enough that a complex and powerful tool such as Refine is not really required to solve it. A simple and equally declarative program in Prolog takes much less time to construct and is only 2 times slower than the equivalent version in Refine. Moreover, the same algorithm implemented directly in Pascal is almost twice as fast as the symmetry-optimized version produce by Refine. So one wouldn't want Refine in order to generate *especially* efficient code.

I think a more helpful conclusion to be drawn is about Refine as a specification language. It is interesting to note that the Costas arrays problem is not trivial to specify in Refine. The week required to formulate the problem is needed because the specification must take into account the kinds of program

3

transformations that Refine was designed to do. In particular, it is necessary to formulate the problem in such a way that distribution laws hold for the functions used in the specification. These appear not to be obvious at first glance.

I conclude that the claim that Refine is a true specification language is a bit suspect. It appeared from the demonstration that given at the CRL that one had to have detailed knowledge of what the refine system was going to *do* to the specification in order to write a specification that generates efficient code. This not very different, it seems to me, from knowing that you should write your Prolog programs in a certain way because your compiler has first argument indexing or an optimization for tail recursion (or knowing that you should avoid Pascal set constructions because they are poorly implemented).

Thus the value of Refine, or any similar tool, probably depends in large part on the programmers ability to understand what the tool is *doing* to assist her in building more efficient programs. This is true also in the comparison of Prolog and Pascal. If the semantics of *freeze* is not clearly understood, the Prolog programs are sufficiently close to one another (the syntactic difference between P2 and P3 is a pair of braces {} and the inversion of the order in a pair of goals) that it may not clear that the entirely different order of execution of goals is responsible for the improvement in performance.

REFERENCES

Smith, Douglas R. "Automating the Development of Software" in the *Proceedings of the fifth conference on Knowledge-Based Software Assistant* Utica, NY, p.13-25, September 1990

van Hentenryck P. *Constraint Satisfaction in Logic Programming*, MIT Press, pp. 224, 1989.

APPENDIX

LP 1
```
%% generate the first N integers into a list Is
integers(N,Is) :- findall(I,integer_range(I,1,N),Is).

%% permutation(List,Permuted).  Permuted is a permuted version of List.
permutation(Xs,[Z|Zs]):- select_element(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).

%% select_element(X,List,Rest).  X is an element removed from List
%% to produce Rest
select_element(X,[X|Xs],Xs).
select_element(X,[Y|Ys],[Y|Zs]) :- select_element(X,Ys,Zs).

%% Length(Integer,List). Generate (or test)
%% that List is of length Integer
length(0,[]) :-!.
length(N,[A|R]) :- successor(M,N), length(M,R).

%% $arg(N,List,Element). Element is the Nth element in List
$arg(1,[Element|_],Element) :- !.
$arg(N,[_|List],Element) :- successor(M,N),$arg(M,List,Element).

%% Two elements are constrained to be different
diff(X,Y) :- { X \== Y }.

%% diff_list(List).  All the elements of List are different
%% if for each element A all the remaining elements R are different
%% ($diff_list) from A
diff_list([_]) :- !.
diff_list([A|R]) :- $diff_list(A,R), diff_list(R).

$diff_list(A,[B]) :- !,diff(A,B).
$diff_list(A,[B|R]) :- diff(A,B), !, $diff_list(A,R).

%% subtract(N,PermutedList,DifferenceList).  For the N-th row of
%% the difference table, an element in that row (DifferenceList) is
%% computed by subtracting the first element from the Nth element
%% of the PermutedList
subtract(N,[E1|R],[D12|S]) :- $arg(N,R,E2),
                D12 is E1 - E2,
                subtract(N,R,S).
subtract(_,_,[]).

%% costas([1,...,N],N,P,[R|CostasArray])
%% given a list of integers [1,...,N], the length of the list N and a
%% permuted list P you create a row R of the costas array
costas([I|Is],N,P,[R|Rs])  :-
   K is N-I,
   length(K,R),              % create a list R of K variables
   diff_list(R),             % constrain them to be different
   subtract(I,P,R),          % compute the elements of R
   costas(Is,N,P,Rs).
costas([_],_,_,[]).
```

```
%% Main Goal
?- N=9, integers(N, L),permutation(L,P), costas(L,N,P,Rs).
```

LP 2

```
generate_diff_lists(N,[_],[]):-!.
generate_diff_lists(N,[I|L],[R|Rs]):-
            K is N-I,
            length(K,R),
            diff_list(R),
            !,
            generate_diff_lists(N,L,Rs).

costas([_],_,_,[]).
costas([I|Is],N,P,[R|Rs])  :-  subtract(I,P,R), costas(Is,N,P,Rs).

%% Same as Costas 1 but with all the diff_lists
?- N=9, integers(N, L),generate_diff_lists(N,L,Rs),
   permutation(L,P), costas(L,N,P,Rs).
```

LP 3

```
subtract(N,[E1|R],[D12|S]) :-  $arg(N,R,E2),
                               {D12 is E1 - E2},   %% freeze arithmetic
                               subtract(N,R,S).
subtract(_,_,[]).

%% Same as Costas 2 with costas/4 test before the permutation/2.
?- N=9, integers(N, L),generate_diff_lists(N,L,Rs),
   costas(L,N,P,Rs),permutation(L,P).
```

P1

```
program main;

   const
      N = 9;          { The size of the Costas arrays we want to generate }
      N1 = N - 1;
      N2 = N - 2;

   type
      differenceTable = array[0..N2] of array[1..N] of -N1..N;
      { The zero's element stores the Costas array itself, the first one}
      { stores the first difference function, and so on. }
                       {      2      4     1      6      5      3     }
                       {            -2     3     -5      1      2     }
                       {                   1     -2     -4      3     }
                       {                         -4     -1     -2     }
                       {                                -3      1     }

   var
      CostasArray: differenceTable;
      Solutions, Ticks: longint;

   function differenceDuplicates (var x: differenceTable;
                                  level, index: integer): boolean;
   { Update the difference table and check for duplicates at all levels.   }
      var
         i, difference: integer;
   begin
      difference := x[0][index - level] - x[0][index]; { Compute the new
                                               difference for that level }
      for i := level + 1 to index - 1 do      {   and check for duplicates }
         if x[level][i] = difference then
            begin
               differenceDuplicates := true;
               exit(differenceDuplicates);
            end;
      differenceDuplicates := false;     { No duplicates found at that level }
      x[level][index] := difference;        { Update the difference fonction }
      if ((level + 1) < index) and (level < N2) then  { If necessary, update
                                            the next difference fonction }
         differenceDuplicates := differenceDuplicates(x, level + 1, index);
   end;

   function duplicates (var x: differenceTable;
                        level, index: integer): boolean;
   { Check for duplicates in the Costas array and, if there are none,
     update the difference table.   }
      var
         i, newElement: integer;
   begin
      newElement := x[0][index];
      for i := 1 to index - 1 do
         if x[0][i] = newElement then
            begin
               duplicates := true;
               exit(duplicates);
            end;
```

```
        duplicates := false;
        if index > 1 then
            duplicates := differenceDuplicates(x, 1, index);
    end;

    procedure generateCostasArray (var x: differenceTable; index: integer);
    { Generate a Costas array by adding an element to the list
      constructed so far and checking for duplicates at each level }
        var
            i: integer;
    begin
        if index <= N then
            for i := 1 to N do
                begin
                    x[0][index] := i;
                    if not duplicates(x, 0, index) then
                        generateCostasArray(x, index + 1);
                end
        else
            Solutions := Solutions + 1;
    end;

begin
    ShowText;
    Solutions := 0;
    Ticks := TickCount;
    generateCostasArray(CostasArray, 1);
    writeln('Found ', Solutions : 1, ' solutions in ', (TickCount - Ticks) div
60 : 1, ' seconds for N = ', N : 1);
end.
```

P2

```
program main;

    const
        N = 9;     { The size of the Costas arrays we want to generate }
        N1 = N - 1;
        N2 = N - 2;

    type
        differenceTable = array[0..N2] of array[1..N] of -N1..N;
        { The zero's element stores the Costas array itself, the first }
        { one stores the first difference function, and so on. }
        {    2        4        1        6          5        3          }
        {        -2        3        -5        1          2        }
        {              1        -2        -4        3        }
        {                  -4        -1        -2        }
        {                      -3        1        }

    var
        Solutions, Ticks: longint;

    function differenceDuplicates (var x: differenceTable;
                                   level, index: integer): boolean;
    { Updates the difference table and checks for duplicates.  }
        var
            i, difference: integer;
```

8

```
begin
   difference := x[0][index - level] - x[0][index]; { Compute the new   }
                                           {  difference for that level }
   for i := level + 1 to index - 1 do     {  and check for duplicates  }
      if x[level][i] = difference then
         begin
            differenceDuplicates := true;
            exit(differenceDuplicates);
         end;
   x[level][index] := difference;    { Update the difference fonction }
   differenceDuplicates := false;    { No duplicates found at that level }
   if ((level + 1) < index) and (level < N2) then  { If necessary, update }
                                        { the next difference fonction }
      differenceDuplicates := differenceDuplicates(x, level + 1, index);
end;

procedure generateCostasArray (var x: differenceTable; index: integer);
{ Adds an element to the permutation constructed so far and checks }
{ for duplicates in the difference table }
   var
      i, temp: integer;
begin
   if index <= N then
      begin
         temp := x[0][index];
         for i := index to N do
            begin
               x[0][index] := x[0][i];
               x[0][i] := temp;
               if not differenceDuplicates(x, 1, index) then
                  generateCostasArray(x, index + 1);
               x[0][i] := x[0][index];
            end;
         x[0][index] := temp;
      end
   else
      Solutions := Solutions + 1;
end;

procedure CostasArray;
{ Takes care of initialization and generate the first }
{ element of the permutation }
   var
      x: differenceTable;
      i: integer;
begin
   Solutions := 0;            { reset solution counter }
   for i := 1 to N do      { initialize Costas array }
      x[0][i] := i;
   for i := 1 to N do    { generate the first element of the permutation }
      begin
         x[0][1] := i;
         x[0][i] := 1;
         generateCostasArray(x, 2); { no checks are needed:
                                      go directly to the next stage }
         x[0][i] := i;
      end;
end;
```

Costas Arrays

```
begin
   ShowText;
   Ticks := TickCount;
   CostasArray;
   writeln('Found ', Solutions : 1, ' solutions in ',
           (TickCount - Ticks) div 60 : 1, ' seconds for N = ', N : 1);
end.
```