

TOWARD EFFICIENCY THROUGH GENERALITY.

Jean Louis LAURIERE
Groupe Structures de l'Information
Institut de Programmation
4 Place Jussieu
75230 Paris Cedex 05 FRANCE

ABSTRACT: In the following lines we shall adduce new elements to a controversy which lies at the heart of current discussions about programming activities. The question is: must we write specific programs or general systems? We shall contrast the two approaches to this methodological problem by means of a case in point. Their respective interest will be compared and a surprising conclusion will eventually be reached which is crucial to the core of AI and justifies many researches in the field: efficiency can only be attained through generality.

I - INTRODUCTION. Work in computer aided programming has two closely related aims: one is program verifying the other is program generating. Since the pioneering results [3], [9] before the seventies, the two basic domains have been extensively studied. Computer aided verifications of programs, assertion-based proofs of programs wherein assertions are both produced and proved automatically, syntactic and semantic transformations of programs, program syntheses from examples, high level languages are the present common topics [2],[8],[10].

In this context, our purpose is on the contrary to show, however strange it might seem, that a specific algorithm, tailored by hand or by program for a particular problem, can be less effective than a general system, the essential nature of which is to be non-deterministic and which is able to deal with quite a large family of different problems.

Thus there is no need for provers or synthesizers but rather for general problem solvers.

This paradoxical proposition will be studied through the enlargement of an example belonging to the field of discrete mathematics. We shall argue over an a priori NP-complete [4] problem:

"Build all the permutations over the first n integers which verify the following constraints, where m and a are two boolean $1 \times (n-1)$ given vectors and σ is whatever solution looked for:

$\forall i \in [1, n-1]: m(i)=1 \Rightarrow \sigma(i) < \sigma(i+1),$
 $a(i)=0 \Rightarrow \forall j \in [i+1, n] \quad \sigma(j) \neq \sigma(i)+1,$
 $a(i)=1 \Rightarrow \forall j \in [1, i-1] \quad ((\sigma(j) \neq \sigma(i)+1) \wedge \sigma(i) \neq n)".$

For example if $n=4$, $m=(101)$, $a=(110)$, the permutations $\sigma_1=(1324)$ and $\sigma_2=(2314)$ are the only two to meet the above requirements.

II - FIRST APPROACH : "the natural one".

The possible solutions must be gradually generated and tested; the classical backtracking scheme is therefore called up and the following "script" is immediately set up in our brain :

- Point α : place the elements one at a time.
- Point β : test the feasibility of the thus obtained partial σ .
- Point γ : carry on if it works, backtrack if part failure or if solution.

This general script must now be made more precise and refined in order to take into account the particulars of our problem. In all cases point α will correspond to something like $i=i+1$ together with $\sigma(i)=\sigma(i)+1$ and with the convenient initializations. As often, point β will now prove a delicate one. Three tests - say t_1 , t_2 and t_3 - must be applied here :

- t_1 : σ is a permutation, i.e. $i \neq j \Rightarrow \sigma(i) \neq \sigma(j)$.
 t_2 : m imposes some monotony constraints.
 t_3 : a imposes less handy constraints.

As in any backtrack approach, the manual optimization task will now consist in setting up these tests early enough and in the right order so as to avoid the generation of useless part solutions. Here, the first test is easy to handle and is certainly the most binding; it will therefore reasonably be undergone first. The monotony test is binding too, and it is much easier to deal with this test than with the conditions derived from a . Indeed this last conditions must be translated by a DO-loop instead of a single IF-statement as with the former ones. Eventually we keep the order t_1 , t_2 , t_3 to obtain our program which is thus nearly

```

completed : i=0 ;  $\sigma \equiv 0$  ;
•  $\alpha$  : i=i+1 ; IF i>n THEN GOTO  $\gamma$  ;
[By hypothesis the first (i-1) elements of  $\sigma$  are
known; a convenient value for  $\sigma(i)$  is looked for]
* :  $\sigma(i)=\sigma(i)+1$  ; IF  $\sigma(i) > n$  THEN GOTO  $\gamma$  ;
    IF i $\neq$ 1 THEN GOTO  $\alpha$  ;
•  $\beta$  : t1 [test on  $\sigma$ ] : DO j=1, i-1 ;
    IF  $\sigma(i)=\sigma(j)$  THEN GOTO * ; END;
    t2 test on m : IF m(i-1)=1 THEN
    IF  $\sigma(i)<\sigma(i-1)$  THEN GOTO * ELSE GOTO t3;
    IF m(i-1)=0 THEN
    IF  $\sigma(i)>\sigma(i-1)$  THEN GOTO  $\gamma$  ELSE GOTO t3;
    t3 [test on a] : ie= $\sigma(i)+1$ ;
    IF a(i)=1 THEN DO; IF ie > n THEN GOTO  $\gamma$ ;
    DO j=1, i-1; IF  $\sigma(j)=ie$  THEN GOTO * ;END;
    GOTO  $\alpha$  ; END;
    IF a(i)=0 THEN DO; IF ie > n THEN GOTO  $\gamma$ ;
    DO j=1, i-1; IF  $\sigma(j)=ie$  THEN GOTO  $\alpha$  ;END;
    GOTO * ; END ;
•  $\gamma$  : [Backtrack]  $\sigma(i)=0$  ; i=i-1 ;
    IF i $\neq$ 0 THEN GOTO  $\alpha$  ELSE STOP;

```

This program has two important virtues: first it was very rapidly written and secondly it works correctly and can easily be proved. However this program can of course be strongly improved. The first way is to find out some good lemmas for the formal problem, the second way is to undertake direct syntactic transformations of the program itself. The article by KNUTH in 1974 [5] originated many researches in such program manipulations. STANDISH et al [11] made up a general catalogue of such accepted transformations. ARSAC [1] programmed a nicely interactive system as an aid to program optimization. We give in [7] a more sophisticated version of the program which takes into account other mathematical properties, which avoids the use of any GOTO statement, and which keeps in memory all the useful information on vector a. The backtrack work will therefore be largely reduced. But we are concerned with a space the size of which is $n!$ and it is likely that any such specialized program cannot work effectively for each couple of m and a. However it seems impossible to achieve much better along this line!

III - SECOND APPROACH : "the general one"

In this section we want to introduce and discuss a quite different way of solving this kind of combinatorial problems. It is in fact clear that all the previous work will prove useless if one should change if only a little the constraints attached to our problem: we can therefore have a mind to write a much more powerful system which will solve this whole family of problems involving discrete finite sets. A great deal of mathematical knowledge and a language to state the different problems are required for achieving this goal.

ALICE, already described in [6], is such a code. The permutation generation problem is given to ALICE in a non procedural manner as follows:

```

GIVEN CONSTANT      n
SETS                 E1=[1,n] ; E2=E1-n.
COEFFICIENTS m      ON E2 ; a ON E2.
FIND BIJECTION        $\sigma$  ON E1 .
WITH  $\forall i \in E2$  m(i)=1  $\Leftrightarrow \sigma(i+1) > \sigma(i)$  AND
 $a(i)=0 \Rightarrow \forall j \in [i+1,n] \sigma(j) \neq \sigma(i)+1$  AND
 $a(i)=1 \Rightarrow \forall j \in [1,i-1] \sigma(j) \neq \sigma(i)+1$  AND  $\sigma(i) \neq n$ .

```

Then the system asks for the missing data : value of n, vectors of coefficients m and a. Alice solves the problem quite in the same manner as we solve it by hand. So the main difference with the previous program is the ability of formal manipulations of mathematical expressions. This allows the system to derive implications, to stick to the numerical characteristics of each problem, therefore to work differently as soon as the data change, to set up good hypotheses and finally to use backtracking in intelligent fashion and non predetermined order. Let us see how ALICE works through some examples of this permutation problem. So if $n=9, m=(11010101)$ and $a=(11110110)$, the system interprets first the constraints deriving from vector m: (A): $\sigma(2) > \sigma(1)$ and (B): $\sigma(3) > \sigma(2)$. Now a zero value, namely m(3) is encountered; so m(i)=1 is false, the program produces : $[\sigma(i+1) > \sigma(i)]$ id est $\sigma(i+1) \leq \sigma(i)$. But as σ is by hypothesis a bijection it obtains finally $\sigma(i+1) < \sigma(i)$; so here with $i=3$: $\sigma(4) < \sigma(3)$ and likewise $\sigma(5) > \sigma(4)$, $\sigma(5) > \sigma(6)$, $\sigma(7) > \sigma(6)$, $\sigma(7) > \sigma(8)$, $\sigma(9) > \sigma(8)$. The constraints derived from vector a give birth to 22 new expressions from $\sigma(1) \neq \sigma(2)+1$ to $\sigma(9) \neq \sigma(8)+1$ and besides the system knows that $\sigma(1), \sigma(2), \sigma(3), \sigma(4), \sigma(6)$ and $\sigma(7)$ cannot be equal to 9. By using an ordinal function the system examines now the constraints all together; $\sigma(1) \leq 6$ and $\sigma(2) \geq 2$ are for instance obtained from (A) and (B). Other values are eliminated in this manner until no other information can be obtained. Now comes the time to choose; the choice will bear on the element which has the fewest possibilities; in this case, our system notices that only two elements can take value 9, these are $\sigma(5)$ and $\sigma(9)$; the tie is broken by considering the element which appears most often in the " $\neq$ " constraints. So setting $\sigma(5)=9$ allows new implications and in particular $\sigma(7) \neq 8, \sigma(6) \neq 7, \sigma(8) \neq 7$. The next choice is $\sigma(3)=8$ and the first solution the system gets to is (678193425). In this case 42 solutions altogether are exhibited. Taking another case with $m=(00001111)$ and $a=(10000111)$ the program obtains 48 constraints, but here it is made obvious from vector m that : $\sigma(1) \geq \sigma(2) \geq \sigma(3) \geq \sigma(4) \geq \sigma(5)$ and therefore as the lowest value of $\sigma(5)$ is one: $\sigma(1) \geq 5, \sigma(2) \geq 4, \sigma(3) \geq 3, \sigma(4) \geq 2$; in the same manner the program

shows that $\sigma(9) \geq 5$, $\sigma(8) \geq 4$, $\sigma(7) \geq 3$, and $\sigma(6) \geq 2$. Now only one element for 5 and one for 9 are convenient: $\sigma(5)=1$ and $\sigma(9)=9$ are made necessary. As $\sigma(5)$ equals 0, $\sigma(6)$, $\sigma(7)$ and $\sigma(8)$ are not equal to 2 and therefore the system comes to $\sigma(4)=2$; in the same way the system deduces $\sigma(7) \neq 3$ and the only convenient antecedent for 3 is 3: $\sigma(3)=3$; then $\sigma(8) \neq 4$ implies $\sigma(2)=4$ which produces $\sigma(6) \neq 5$. Substitutions give: $\sigma(1)=5$, $\sigma(6)=6$, $\sigma(7)=7$, $\sigma(8)=8$ in that order the unique solution has been reached directly. We notice that the problem solver uses, by discovering it implicitly, the fact that $\sigma(j)=n$ can only occur for a j such that $m(j-1)=1$, $m(j)=0$ and $a(j)=0$; such a lemma is very useful as an aid for fruitful choices. Many other results are in fact found by the system as it proceeds through formal manipulation of constraints. (This permutation problem can be viewed too as a E1 onto itself matching problem and some convenient graph procedures allow the system to derive other implications through graphical arguments [6]).

Let us take a last example to see an obviously polynomial with n behaviour of ALICE together with a catastrophic execution of the specific program first written. Suppose for $n=50$ or greater we set: $m(1)=a(1)=1$ and for any $i > 1$: $m(i)=a(i)=0$. In ALICE the constraints are interpreted and produce $\sigma(1) < \sigma(2)$ with $\sigma(2) > \sigma(3) > \sigma(4) > \dots > \sigma(n)$; thus $\sigma(2)=n$ and $(n-1)$ values remain for $\sigma(1)$; since none contradicts the hypotheses on vector a , $(n-1)$ solutions are immediately printed: $(n-1, n, n-2, n-3, \dots, 1), \dots, (1, n, n-1, n-2, \dots, 2)$. On the contrary, the execution of the first written program is not good at all; it will generate the partial permutations 12, 132, ..., 8 9 4 1, ..., 8 9 4 3 2 1, ... with many and many useless attempts. Furthermore, in all of the previous cases, additional constraints like: $\sigma(1)=n/2$, $\sigma(n/2)=1$ or others, will but make the task of our general problem solver easier, as proved on machine, while in the first approach, one is forced to build new and new lemmas again and to design in each case a new specialized program.

IV - CONCLUSION.

My aim was not to claim that it is impossible to write a specific program which can work correctly on specific examples; but, there are always new particular cases to take into account and therefore some new adequate statements to add into this program. So, little by little, one is induced to write a quite large code; at this point, if we try a bit harder, we can deal with many other combinatorial problems, say for instance the eight queens, cryptarithmic, integer mathematical programming, optimization problems and so on: such is indeed the scope of a general program. It has been actually shown by ALICE, in all these cases, that generality does

not contradict efficiency: numerical and graphical analysis, supported by good heuristics adjusted to each particular case, allows the system to foresee quickly constraint violation even with large size problems.

- Why can a specific program work badly? The first reason is that it lacks one indispensable tool: formal manipulation; although we use this tool very commonly by hand, too few specialized programs are able to do so. The second, perhaps more important, reason has to do with the trivial statement $i=i+1$ just at the beginning of our specific backtrack programs; this means that we construct the objects we are looking for in a predetermined order. But this is unfortunately not the right way to proceed: one element may have an imposed value or another may have significantly fewer possibilities than the others. In one word when we write a specific program we make implicit choices: we have to decide without correct elements of decision because the missing information lies in the data. In that way, to each specific algorithm will correspond a worst case. On the opposite this can hardly be said of a general system with good mathematical knowledge together with heuristics based on rich and varied representations of objects in the universe dealt with: it discovers facts about the problem at the problem statement level not at the code level.

• REFERENCES

- [1] Arzac J.: "Syntactic source to source transforms and program manipulation". CACM 22,1(1979) 43-54.
- [2] Burstall R. and Darlington J. "A system which automatically improves programs" Acta Informatica 6, (1976), 41-60.
- [3] Hoare C.: "An axiomatic basis for computer programming" CACM 12,10(1969), 576-583.
- [4] Karp R.: "On the computational complexity of combinatorial problems" Networks 5, (1975), 45-68.
- [5] Knuth D.: "Structured programming with GOTO statements" ACM Comp. sur. 6,4(1974), 261-301.
- [6] Laurière J.L.: "A language and a program for stating and solving combinatorial problems" Art. Int. 10, 1 (1978), 29-127.
- [7] Laurière J.L.: "Specific algorithms versus general programs", GR 22, A.I. (1979).
- [8] Lee et al.: "An improved program synthesis and its correctness" CACM 17,4(1974), 211-217.
- [9] Manna Z.: "The correctness of programs". J. Comp. Sys. Sciences 3,2(1969), 142-156.
- [10] Manna Z., Waldinger R.: "A deductive approach to program synthesis" SRI Tech. R.177 (1978).
- [11] Standish et al.: "Improving and refining programs by program manipulation". ACM Nat. C. (1976), 509-516.