From: AAAI-88 Proceedings. Copyright ©1988, AAAI (www.aaai.org). All rights reserved.

Generality versus Specificity: an Experience with AI and OR techniques

Pascal Van Hentenryck
ECRC
Arabellastr. 17,
8000 Muenchen (F.R.G)

Jean-Philippe Carillon
CEGOS
204, Rond-point du Pont-de-Sevres
92516 Boulogne-Billancourt (France)

Abstract

This paper contains an in-depth study of a particular problem in order to evaluate several approaches to the solving of discrete combinatorial problems. We take a warehouse location problem as a case study and present solutions to it by using Integer Programming, a specialized program based on A^* and the constraint logic programming CHIP. The merits of each approach are discussed and compared in the light of the problem. Finally, we conclude by arguing that CHIP provides a valuable addition to the current set of tools for solving discrete combinatorial problems.

1 Introduction

Many real world problems in Artificial Intelligence, Operations Research (OR), VLSI design and Computer Science in general can be viewed as Constraint Satisfaction Problems or discrete combinatorial problems. Because of the NP-complete nature of these problems, no efficient general algorithm is available for solving all of them. It follows that this class of problems implies a trade-off between generality and efficiency. Current approaches to the solving of discrete combinatorial problems can be essentially classified into three opposite groups

- the use of general tools like Integer Programming packages or theorem provers.
- the writing of specialized programs in procedural languages.
- the use of a high-level declarative language embedding some advanced AI techniques.

The main advantage of general tools is their wide applicability. Most problems can be easily expressed in their problem-solving model. However their efficiency depends upon the nature of this expression. Whenever a problem can be expressed naturally in their model (e.g. the mathematical model of Integer Programming [Garfinkel and Nemhauser, 1972]), this approach is very effective. But for most problems, a *recasting* operation takes place which can substantially increase the number of variables and hence the search space to explore and hide some of the problem features (e.g. symmetries, heuristics, ...) preventing the general tools from exploiting them.

The writing of specialized programs in procedural languages is the dual of the first approach. Here the accent is put on efficiency by exploiting as much as possible of the problem features. This is likely to be the most efficient approach for many problems. Unfortunately, the development time of these programs is significant; also they are generally rather inflexible as it requires much programming to change or to extend them.

The third approach, the use of a high level declarative language, tries to preserve as much of the efficiency of the second approach while reducing substantially the development time and increasing the flexibility of the programs. Of particular interest is the constraint logic programming language CHIP which has been developed with precisely this idea in mind. CHIP combines the declarative aspects of PROLOG with the efficiency of constraint handling techniques [Dincbas *et al.*, 1987]. The constraint handling part of CHIP includes *Consistency Techniques* [Van Hentenryck, 1987b], an important paradigm emerging from Artificial Intelligence (e.g. [Mackworth, 1977]). Other high-level constraint languages include CONSTRAINTS [Sussman and Steele, 1980], CLP(R) [Jaffar and Lassez, 1987] and Prolog III [Colmerauer, 1987].

This paper presents an in-depth analysis of these approaches on a particular problem from OR. The *warehouses location problem* (section 2) is taken as a case study and we present solutions to it by using one representative of each class, respectively Integer Programming (section 3), a specialized program (section 4) and CHIP (section 5). We then conclude by discussing the merits of each approach and providing a more general perspective on the solving of discrete combinatorial problems (section 6).

2 A warehouses location problem

Assume that a factory has to deliver goods to its customers and has at its disposal a finite number of locations where it is possible to build warehouses. For each warehouse, we have a cost, referred to as a fixed cost, representing the construction and the maintenance of this warehouse. There are also variable costs for the transportation of goods to the customers. These costs are variable since they are dependent on the warehouses locations (because of the distance). The problem is to determine the number and the locations of the warehouses which minimize the (fixed and variable) costs. In the rest of this paper, we take the following naming conventions

- m: the number of warehouse locations;
- n: the number of customers;
- b_j: the demand in goods of customer j;
- f_i: the fixed cost of warehouse i;
- c_{ij}: the unit transport cost from warehouse i to customer j;

- $v_{ij} : c_{ij}\, b_j$.

The above statement is a particular case of the warehouses location problem where no capacity constraints are enforced on the warehouses. Although simpler (but still NP-Complete) than the general problem, it fully serves the purpose of this paper.

3 Integer Programming

3.1 Problem solution

In this section, we show how the above problem can be formulated as an Integer Programming problem. This requires stating the problem in terms of integer variables and linear equations or inequations. For this purpose, we introduce the following variables

- w_i which is equal to 1 if warehouse i is open and to 0 otherwise.
- g_{ij} which is equal to 1 if warehouse i delivers goods to customer j and to 0 otherwise.

Now the problem can be stated as follows

$$min\, z = \sum_{i=1}^{m}\sum_{j=1}^{n} v_{ij}\, g_{ij} + \sum_{i=1}^{m} f_i w_i$$

subject to

$$\sum_{i=1}^{m} g_{ij} = 1 \; (j = 1, \ldots, n)$$

$$\sum_{j=1}^{n} g_{ij} \leq n\, y_i \; (i = 1, \ldots, m)$$

$$g_{ij},\ w_i = 0 \text{ or } 1 \; (j = 1, \ldots, n \text{ and } i = 1, \ldots, m)$$

This program can now be solved with standard algorithms such as branch & bound or cutting plane methods [Garfinkel and Nemhauser, 1972].

3.2 Evaluation of the approach

This approach clearly does not require much effort if one has a Integer Programming package at his disposal. However, it is likely to be very inefficient. Note that the number of variables in this formulation is $nm + m$. The search space to explore is thus 2^{nm+m}. In case of real life problems (e.g. 20 locations and 80 customers), this gives rise to huge Integer Programming problems.

The problem with this formulation is that we lose the privileged role played by the variables w_i. As shown in the next section, only these variables matter. But Integer Programming has no way to deduce it from the formulation. Similarly, it is not possible to express heuristics for the choice of the open warehouses. This illustrates a common fact about Integer Programming. Most of the time, the problem structure is lost, entailing much redundancies during the search.

4 A specialized program

4.1 Problem solution

We now present a specialized program for the warehouses location problem which has been developed by the second author of this paper and turned into a software product (running on a micro-computer) that was instrumental in solving real world problems. It implements a specific branch & bound algorithm but can also be viewed as an implementation of the A^* procedure [Nilsson, 1982].

The basic idea behind the program is to reason about the warehouses. Once the number and the location of the warehouses have been chosen, it is a simple matter to assign each customer to a warehouse; we simply choose the closest one which is open. Therefore the search space to explore is the set of all possible warehouse configurations.

A node in the branch & bound will represent a set of configurations. Of course, this set is not represented explicitly but can be characterized by mentioning which warehouses are always open or closed in these configurations and which are still undecided.

To fully specify the program, we have to define

the branching process: the way a node is split into sub-nodes;

the search rule: the way the next node to work on is selected.

the bounding process: the way the evaluation of the node is carried out;

The branching process is achieved by fixing the value for a still undecided warehouse. From a particular node, two sub-nodes are generated, one where the warehouse is open and one where it is closed.

The search rule selected in the specialized program was to select first the node with the best evaluation. In OR terminology, this means that the program conducts a *best-first* branch & bound.

The bounding process amounts to find an evaluation of the best configuration available from a node. At this node, the best possible value for the variable cost of a particular customer is when this customer receives goods from the closest warehouse which is open or undecided. We refer this value as the best potential cost for this customer at that node. Now the evaluation of a node can be defineds as the summation of all the fixed costs of the open warehouses plus the summation of the best potential costs for the customers. It can be seen that this evaluation is optimistic; it is always smaller or equal to the value of the best configuration. Indeed, "opening a warehouse" increases the evaluation since it adds a fixed cost and "closing a warehouse" may increase the evaluation since it may raise the best potential cost of some customers. The optimistic nature of the evaluation makes possible to rule out a node as soon as its evaluation is greater than the value of an already found solution.

Note that better evaluation functions can be found for this problem if we measure efficiency by the number of generated nodes. However, it was found while experimenting with real world problems that such functions increase the CPU time needed for solving them.

4.2 Evaluation of the approach

The search space to explore is 2^m within this approach. This is to contrast with the Integer Programming formulation. Specific heuristics (e.g. the choice of the warehouse on which is based the branching process) can also be included inside the algorithm. Therefore, as far as efficiency

is considered, this approach is fully appropriate, simply because it takes the problem features into account.

When ease of programming is considered, things are just the other way around. The program kernel consists in about 2000 lines of Pascal and requires several months of development time. Equally important is the fact that this approach is rather unflexible. Changing the heuristics or adding new constraints would imply an important programming task. For instance, adding a disjunctive constraint between two warehouses (i.e. at most one of them can be selected in a solution) requires a complex change of the system. Adding a capacity constraint on the warehouse would require a complete redesign of the system.

5 A CHIP solution

We now present a solution of the warehouses location problem in the constraint logic programming language CHIP. This presentation is essentially self-contained but more details about CHIP can be found in the given references.

5.1 Problem solution

The program is presented by successive refinements, starting with a first naive program and then adding more features to make it more efficient.

Finding solutions. We first define a program for yielding solutions. The basic approach for doing so in CHIP consists in defining a program which generates the problem constraints and a program which generates values for the problem variables, i.e.

```
location(Lware,Lcust,Cost) :-
    gen_constraint(Lware,Lcust,Cost),
    gen_value(Lware,Lcust).
```

The first predicate in the body defines two lists of domain-variables and sets up the constraints between these variables. **Domain variables** [Van Hentenryck and Dincbas, 1986] is one of the main extensions of CHIP compared with usual logic languages. They are similar to logic variables except that they can only take a finite set of values. Domain variables are the basic extension for embedding Consistency Techniques in logic programming [Van Hentenryck, 1987a]. During the constraint propagation, domains are reduced, possibly leading to instantiations of variables (when only one value remains) or to a failure (when no value is left). This step is studied in more details in the next sections.

The second predicate makes choices for the variables. The choice process can be very different from one program to another, e.g. it might be based on *instantiation* or *domain-splitting* [Van Hentenryck, 1987b]. In the present program, it can simply be defined as follows.

```
gen_value(Lware,Lcust) :-
    labeling(Lware),
    labeling(Lcust).

labeling([]).
labeling([X|Y]) :-
    indomain(X),
    labeling(Y).
```

We first start by choosing which warehouses are included in the solution and then we assign a warehouse to each customer. The `indomain` predicate simply generates values for the variables. It gives to the variable a value from its domain. If backtracking occurs at this point, another value is tried and so on until none are available. Note that all the tree-search is abstracted away inside this predicate.

During execution, these two steps work as coroutines. The constraint propagation is started by the first step. When no more information can be deduced, a choice is made in the second step. This brings additional information, which may restart constraint propagation. Note that this mechanism has not to be programmed by the user. This is provided directly by CHIP.

Defining the variables. We herein consider the variables used inside the program.

`Lware` is a list of zero-one domain-variables for the warehouse locations. The i^{th} such variable w_i will be assigned to 1 if the i^{th} warehouse location is open in the solution and will be zero otherwise.

`Lcust` is a list of domain-variables ranging over [1,n] for the customers. The variable g_j represents the warehouse which delivers goods to the j^{th} customer.

We also use another domain-variable vc_j for each customer. The variable vc_j represents the variable cost associated with customer j. This variable ranges over the possible variable costs associated to this customer.

Finally, the evaluation function can be defined as

$$f_1 \times w_1 + \ldots + f_m \times w_m + vc_1 + \ldots + vc_n.$$

Relating the customers and their variable costs. We now have to define the constraints enforcing the relations between these variables.

We need a constraint to make the correspondence between the values of the variables g_j and the variables vc_j ($1 \leq j \leq n$). This is achieved through the predicate

```
element(I,L,El)
```

`element(I,L,El)` holds iff the I^{th} element of L is El. This predicate is handled in a looking ahead way. For the j^{th} customer, the constraint looks like as follows

$$\texttt{element}(g_j, [v_{1j}, \ldots, v_{kj}, \ldots, v_{mj}], vc_j).$$

with v_{ij} defined as in section 2.

There are n such constraints since they are n customers. The pruning achieved by this constraint can be described in the following way. As soon as the cost vc_j is updated, some now inconsistent values of g_j are removed from consideration. Similarly, when g_j is updated (e.g. a value is removed from its domain), the cost is updated in correspondence.

Symbolic constraints such as the `element` constraint are essential tools for solving many discrete combinatorial problems. Part of the CHIP efficiency comes from the ability to handle them. They enable the programs to be stated and solved in a natural form even if some constraints are not linear as it is the case in the present example. In principle, any symbolic constraint which can be expressed as a logic program can be handled in CHIP.

Relating the warehouses and the customers. There remains one constraint to be expressed to conclude the description. This constraint simply states that, whenever a warehouse is closed in the solution, no customer can receive goods from it.

Such constraint could be taken into account during the generation step. However doing so will lead to a less declarative program. If, in the future, we want to change the generator, we will have to take care about this constraint. A better way is to separate the constraint from the generator and to coroutine their execution.

We define a predicate `removefromcustomer(Wa,Cu)` which, given a list Wa of 0 or 1 and a list Cu of integers, holds if all the elements of Cu are different of i if the i^{th} element of the list Wa is 0. In the program, the first argument stands for the list of warehouse locations and the second one for the list of customers.

A simple definition can be

```
removefromcustomer([X|Y],Lcust) :-
  removefromcustomer([X|Y],Lcust,1).

removefromcustomer([],_,_).
removefromcustomer([X|Y],Lcust,Nb) :-
   ifoutof(X,Nb,Lcust),
   Nb1 is Nb + 1,
   removefromcustomer(Y,Lcust,Nb1).

ifoutof(0,Nb,Lcust) :-
   outof(Nb,Lcust).
ifoutof(1,Nb,Lcust).
```

Adding the declaration

```
delay ifoutof(ground,any,any).
```

will coroutine its execution with the generator. Indeed, the above program will generate a set of `ifoutof` constraints. Such a constraint cannot be selected until the first argument becomes ground. When this happens, the constraint is selected and removes a warehouse location from the possible choices for the customer if the warehouse has been "closed". The ability to separate the definition of the constraint (the logic) and the way to use it (the control) is responsible for the ease of programming and the great modifiability of CHIP programs.

Finding the optimal solution. So far, we have a program to generate solutions. To find the optimal solution, we use the higher-order `minimize(G,F)` where G is a goal and F a linear expression. This meta-level predicate solves the goal G in a way that minimizes the value of F. It implements a depth-first branch & bound technique [Van Hentenryck, 1987b]. More precisely, this predicate will search for a solution of Goal. Once such a solution has been found with a cost C (i.e. the value of F for this solution), a constraint $F < C$ is dynamically generated which constrains the search for the other solutions. The process terminates when all the search space has been implicitly searched through. The handling of the generated constraints is achieved through a reasoning on the variation intervals, similar in essence to the one of [?]. The program now looks like as follows

```
location(Lware,Lcust,Cost) :-
    gen_constraint(Lware,Lcust,Cost),
    minimize(gen_value(Lware,Lcust),Cost).
```

Let us have a look at how are evaluated the partial solutions with this program. Remember that the expression to minimize is

$$f_1 \times w_1 + ... + f_m \times w_m + vc_1 + ... + vc_n.$$

Suppose we have already assigned values to some warehouses. It follows that some w_j have already received a value. Also, if some w_j have been assigned to 0, some of the values in the domains of vc_i can have been removed. The system evaluates this expression in an optimistic way by assuming that each remaining domain-variable gets its smallest possible value. This is equivalent to the summation of

1. all the fixed costs of the warehouse locations that have be retained in the partial solution
2. the best potential costs for the customers.

It follows that the natural statement of this problem in CHIP directly leads to the same bounding process as in the specialized program.

Now we already have a working program. Note that we never specify that the customers must receive their goods from the closest warehouse. This fact is discovered independently by the program. If we change the generator for the customers so that it assigns first the closest possible warehouse, the program will never consider any other choice, the evaluation function pruning them automatically. Hence it yields the same complexity result as the specialized program. We now show how to improve the efficiency of the program by adding some redundant constraints and/or searching for a good first solution.

Adding redundant constraints. There are some warehouse locations that are not worth considering for a given customer because the construction of another warehouse location will induce a smaller cost. A simple logic program can be written to enforce these constraints.

Searching for a first solution. It is generally a good idea to search for a good solution before starting the branch & bound. This helps pruning the search space since we will only look at solutions with a smaller cost. Several strategies can be used for this step. The one we retain was

- Trying to minimize the number of warehouses.
- Ordering the warehouses in function of their proximity to the customers.

Once we have computed a good first solution, we search for the optimal solution and prove optimality by using a dual approach for generating values, trying first to have the greatest possible number of warehouses.

Sketch of the final program. So the final program looks like the following

```
location(Lware,Lcust,Cost) :-
    defining(Lware,Lcust,Cost),
    removeredundant(Lcust,Lware),
    searchinggoodsol(Upper),
    minimize(gen_value(Lware,Lcust),Cost,Upper).
```

The alterations of the basic program are the predicates for enforcing the additional constraint and for searching a good solution. The latter is similar to the basic scheme except that it uses another generator of values. Finally, the higher-order predicate for optimization has one more argument, the upper bound computed during the search for a good solution. This means that we are only searching for solutions whose cost is smaller than this upper bound.

5.2 Evaluation of the approach

In terms of efficiency, the above program is comparable to the second approach. Real-life problems have been solved within a few minutes. For instance, the optimal solution of a 21-20 instance is found and proved optimal in 20 seconds on a SUN 3/160 with our prototype interpreter. A similar result is obtained in 90 seconds for a 21-80 instance. Most of the second approach's efficiency is preserved, the results being about 5 to 10 slower than the specific program.

The fundamental advantage of the third approach comes from its flexibility. The overall program is two pages long and a few days were required to understand the problem and to write it. Changing the program is not a difficult matter. It is straightforward to add a disjunctive constraint between two warehouses. We simply add a linear inequation in the first step. In the same way, it would take a couple of hours to include capacity constraints; it mainly amounts to remove the redundant constraint and to change the heuristics for finding the first solution.

6 Discussion

This paper has presented an in-depth analysis of a warehouses location problem as a case-study for evaluating several approaches to the solving of discrete combinatorial problems. Two of them have been fully implemented and experience with both programs has been reported. How can we summarize this experience ? As far as convenience of programming is considered, Integer Programming turns out to be the ideal solution. Unfortunately, the inability to exploit the problem features makes it very inefficient. The other two approaches provide realistic approaches for solving real world problems. Although the second approach turns out to be slightly more efficient, the lost in efficiency of the third approach is largely compensated by its short development time and flexibility.

How can we generalize this experience ? None of these approaches is adequate for all problems. Therefore it is interesting to identify the problems and the approach that applies.

As far as the problem can be viewed naturally as an Integer Programming problem, the first approach is the way to go. It provides both efficiency and ease of programming.

The third approach, the use of CHIP is appropriate for all problems where it is difficult to extract or to exploit mathematical properties. CHIP is then an efficient and flexible way to solve the problem. Its efficiency comes from Consistency Techniques, the ability to take into account problem features and the handling of symbolic constraints. In addition, it makes possible to write extensible and flexible programs in a rather short time. Note that the ease of programming can directly influence the efficiency. People are likely to exploit some problem features which would require otherwise much programming effort. As such, CHIP is also a valuable prototyping tool. Many real world problems are in its problem-solving scope. For instance, CHIP has been applied successfully to graph coloring, disjunctive scheduling, two-dimensional cutting-stock problems, assembly-line scheduling, microcode labeling problems and channel routing. Some of these applications can be found in [Van Hentenryck, 1987b]. For all these problems, CHIP is comparable in efficiency with specialized programs.

Counterexamples to this class are, for instance, travelling salesman and transport problems. Their mathematical properties enable, for instance, powerful relaxation methods to be used. Therefore the natural formulation of these problems within CHIP will not be able to compete with specialized programs based on these techniques. It is of course always possible to write programs exploiting these properties in CHIP but we then lose its basic advantages. For these kinds of problems, it is clear that the second approach is the most appropriate.

It follows from this analysis that all three approaches are complementary for solving real world discrete combinatorial problems and that CHIP is a valuable addition to the set of conventional tools.

Acknowledgments.

The first author gratefully thanks Mehmet Dincbas, Herve Gallaire, Alexander Herold, Helmut Simonis and the members of the CHIP group for numerous discussions.

References

[Colmerauer, 1987] A. Colmerauer. Opening the Prolog-III Universe. *BYTE Magazine*, 12(9), August 1987.

[Dincbas *et al.*, 1987] M. Dincbas, H. Simonis, and Van Hentenryck P. Extending Equation Solving and Constraint Handling in Logic Programming. In *CREAS*, Texas, May 1987.

[Garfinkel and Nemhauser, 1972] R.S Garfinkel and G.L Nemhauser. *Integer Programming.* John Wiley & Sons, 1972.

[Jaffar and Lassez, 1987] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich (FRG), January 1987.

[Mackworth, 1977] A.K. Mackworth. Consistency in Networks of Relations. *AI Journal*, 8(1):99–118, 1977.

[Nilsson, 1982] Nils Nilsson. *Principles of Artificial Intelligence.* Springer-Verlag, 1982.

[Sussman and Steele, 1980] G.J. Sussman and G.L. Steele. CONSTRAINTS-A Language for Expressing Almost-Hierarchical Descriptions. *AI Journal*, 14(1), 1980.

[Van Hentenryck, 1987a] P. Van Hentenryck. A Framework for Consistency Techniques in Logic Programming. In *IJCAI-87*, Milan, Italy, August 1987.

[Van Hentenryck, 1987b] P. Van Hentenryck. *Consistency Techniques in Logic Programming.* PhD thesis, University of Namur (Belgium), July 1987.

[Van Hentenryck and Dincbas, 1986] P. Van Hentenryck and M. Dincbas. Domains in Logic Programming. In *AAAI-86*, Philadelphia, USA, August 1986.