Constraint Programming versus Mathematical Programming

Jesper Hansen*

February 7, 2003

1 Introduction

Constraint Logic Programming (CLP) is a relatively new technique from the 80's with origins in Computer Science and Artificial Intelligence. Lately, much research have been focused on ways of using CLP within the paradigm of Operations Research (OR) and vice versa. The purpose of this paper is to discuss the differences and similarities between the two mentioned paradigms and how they can complement each other.

In OR and Mathematical Programming we are modeling and solving hard combinatorial optimization problems with use of Linear Mixed Integer Programming models (MIP). *Programming* in MIP has nothing to do with *computer programming*. Rather, the result is a program or schedule for doing the optimized activities. In CLP on the other hand a constraint program is both a statement of a problem in variables and constraints and a specification of how to solve it. A CLP system contains a programming language component and a *constraint store* component that can store and "reason" about variables and constraints.

In the OR community we are interested in formulating and solving models of the following type:

$$\min z = \sum_{j=1}^{m} c_j x_j \tag{1}$$

$$\sum_{j=1}^{m} a_{ij} x_j = b_j \quad i = 1, ..., n \tag{2}$$

$$x_j \geq 0 \quad j = 1, ..., m \tag{3}$$

$$x_j \text{ integer} \quad j = 1, ..., m \tag{4}$$

A wide range of applications can be modeled and solved within this paradigm, but some are not well suited. The objective function and the constraints have to be formulated as linear functions[1], which is not always simple to do.

CLP on the other hand can handle non-linear functions in any type of constraint, for example $x^2 \neq y$. The increase in expressiveness when modeling is the major motivation to take a closer look at CLP from an Operations Researcher's point of view.

*Informatics and Mathematical Modelling, Technical University of Denmark, 2800 Lyngby, Denmark, email:jha@imm.dtu.dk

[1]Almost ignoring quadratic, semi-definite and other convex models solvable with use of interior point methods.

The CLP system *ECLiPSe* is a very flexible environment for implementing and solving both MIP, CLP and combined models, which we will use in the paper to illustrate the different possibilities. ECLiPSe is based on Prolog, but we will only describe specific Prolog constructs, which are needed for our purposes. Clocksin and Mellish [2] is the classic, but perhaps a bit out-dated book on Prolog.

Before considering the modeling possibilities in CLP we shall in section 2 recall the components of Branch & Bound for solving MIP problems and compare that to the CLP approach. In section 3 we show an example of modelling and solving a problem with both a MIP and a CLP model in ECLiPSe.

The potential of both paradigms is even more evident when they are combined into *hybrid solvers*, which is however not covered in this article.

2 Comparing Branch & Bound and CLP

Assume we are to minimize the objective function, $f(x)$ given by (1) over the region of feasible solutions, $x \in S$ given by equations (2)-(4). Let z^{opt} be the optimum value.

2.1 Branching and Tree Search

In order to guarantee finding the best solution in Branch & Bound, all solutions need to be enumerated in the worst case. This is basically done by traversing a search tree. In every node of the tree we divide the solution space into smaller subproblems, which are easier to solve. One way is to select a variable and in each subproblem or branch of the tree fix the variable to each of the values in its domain. Another approach is to split the domain in two branches by adding the constraint $x \leq v$ to the left branch and $x \geq v + 1$ to the right for some value $v \in D_x$, where D_x is the domain of the variable x.

For small problem instances as for instance 10 binary variables, we get 1024 potential solutions or leaves in the search tree. Increasing the number of variables to only 100 binary variables, we get a 31 digit number of potential solutions. This exponential growth makes it impossible in practice to do an explicit enumeration of all solutions for problems of an interesting size.

Search by branching is the most obvious similarity between Branch & Bound and CLP.

2.2 Bounding by Relaxation

Branch & Bound is an implicit enumeration as it uses a bounding function to prune the search tree and only consider parts of the tree implicitly. A bounding function is found by relaxing some set of complicating constraints making the relaxed problem easy to solve. Let P be the region of solutions defined by the relaxed problem. Optimizing the objective function over P will give a lower bound z^{LB} on z^{opt}, since we are optimizing over a larger region.

Assume that the best solution found so far has value z^{best}. Now there are three cases where we can prune the search tree:

- $z^{LB} \geq z^{best}$ – no improving solutions in this subproblem.
- No feasible solutions can be found for the relaxed problem, and hence not for the original problem.
- A feasible solution is found to the original problem.

Bounding is not a part of the CLP paradigm. Instead optimization can be done in the following way: Each time a feasible solution has been found with objective value z^{best} a new constraint is imposed, $z < z^{best}$ making only improving solutions feasible.

The most widely used relaxation is *LP-relaxation* where the integrality constraints of the problem are relaxed resulting in a LP problem.

2.3 Strengthening by domain reduction and cuts

A very powerful extension of Branch & Bound is the possibility of adding *cuts* or constraints during the search to strengthen the relaxation. By finding and adding general or problem specific cuts, we want to remove parts of the relaxed solution space and ideally achieve the convex hull of the feasible integer points. This procedure is referred to as *Branch & Cut.*

In Branch & Bound the tree can be pruned, if no feasible solution exists for the relaxed problem in the current node. In CLP infeasibility is detected when any variable gets an empty domain. The variable domains are reduced by use of *constraint propagation.* Consider the constraint $x_1 \geq x_2 + 1$ for $x_1, x_2 \in \{0, 1, 2\}$. No feasible value for x_1 exist when $x_2 = 2$. The value 2 can hence be removed from the domain of x_2. Similarly is the value 0 in the domain of x_1 inconsistent with any remaining values of x_2. At some stage in the search tree we would branch on x_1 and fix it to 2. Then the value 2 is not consistent in the domain of x_2 and it is removed and x_2 is fixed to 1. This is the most simple example of domain reduction, resulting in removal of the branching on x_2.

The reduced domains of x_1 and x_2 are propagated to other constraints including x_1 and x_2, which can again cause new domain reductions. This procedure continues until no further reductions can be performed and the resulting state is called consistent. Now 3 cases can occur:

1. All variables have exactly one value in its domain and a feasible solution is achieved.
2. One or more variables has no values in its domain – no feasible solutions.
3. At least one variable has a domain size larger than one.

In case of 1 and 2 the search backtracks or the branch is pruned in Branch & Bound terms. In case 3 a variable with domain size larger than one is selected and branching continues.

Generally, a value can be removed from the domain of a variable, if the value is inconsistent with any combination of possible values of the other variables in the considered constraint. If all values in the domains of the variables are consistent, the constraint is called *consistent.*

There are different types of consistency:

Node consistency: The constraint only includes one variable.

Arc consistency: The constraint includes two variables.

Hyper-arc consistency: The constraint has more than 2 variables. In general it is $\mathcal{NP}$-hard to show that a constraint is hyper-arc consistent and hence as hard as solving an arbitrary satisfiability problem [4].

In practice, hyper-arc consistency is not used. Instead the concept of *bounds consistency* is introduced. Consider the constraint $x = y + z$. Let $\min(D_x)$ and $\max(D_x)$ denote the minimum and maximum value in the domain of the variable x. The bounds on x can now be reduced by using the following constraints:

$$x \geq \min(D_y) + \min(D_z), \quad x \leq \max(D_y) + \max(D_z)$$

Isolating the other variables in the forms $y = x - z$ and $z = x - y$ we can reduce the domains of y and z in the same way - here shown for y:

$$y \geq \min(D_x) - \max(D_z), \quad y \leq \max(D_x) - \min(D_z)$$

Algorithms for reducing domains usually only consider one constraint at a time, but in many cases more reduction could be achieved by considering more than one constraint at a time. There is however a clear dependency between the complexity of the algorithm and the ability to reduce the domains. The choice of which algorithm to choose to get the best quality/time compromise can be problem dependent.

Domain reduction is not new in the OR community, since it has been a main part of specially tailored Branch & Bound algorithms. Domain reduction is as well used in presolvers of LP and MIP solvers, but only in the root of the search tree as the name also indicates. Note that from the point of view of Branch & Cut, domain reduction can be seen as adding simple cuts removing parts of the domain.

2.4 Designing Branch & Bound

In a combined framework of Branch & Bound and CLP the following design issues must be considered:

Bounding method: Choice of relaxation and bounding function.

Cuts and propagation: Choice of cuts to add as well as propagation strategies and algorithms.

Selection of next variable to branch on: Often one can do better than to branch on a random variable. In binary branching picking a variable where fixing it to one value gives a low bound and to the other a high bound. Hopefully, this will lead to a good feasible solution in the left subtree and make it possible to prune the right branch. In CLP we generally want to branch on "important" variables first. For a given problem it is often quite obvious, which variables are the most important. Later we will se an example on this. Generally, branching on variables with a small domain size first is a good choice for minimizing the number of nodes in the tree.

Branching strategy: When subdividing the solution space, instead of branching on all values in the domain of a variable, which is common in CLP, the LP-relaxation can be used to split the domain into two. Let $\overline{x}$ be the non-integral value of the LP-relaxation. The problem is then split into two subproblems given by the two constraints: $x \leq \lfloor \overline{x} \rfloor$ and $x \geq \lceil \overline{x} \rceil$.

The following branching example is often used when solving scheduling problems. Here the decision must be taken whether activity i should be executed before j or j before i. This can be embedded in the branching strategy. In one branch the constraint $t_i + p_i \leq t_j$ is added, where t is the start time and p the duration, and in the other branch the constraint $t_j + p_j \leq t_i$. This strategy is not available in a standard MIP solver.

The order in which values are tried in the usual CLP branching is often important. Perhaps it is known in advance that values in the middle of the domain are most likely to lead to good solutions. Incorporation of problem specific knowledge is necessary to make the search efficient.

Search strategy: Basically 3 pure strategies can be identified. In *Depth first search* a subtree is fully explored before backtracking, while in *Best first search* we always explore the most promising node first. Finally in *Breath first search* all nodes in the same level of the tree are explored before continuing to the next level. The advantage of Depth first search is the limited memory consumption, since the number of active nodes will be proportional to the maximum level of the tree. It is as well focused on finding a feasible solution as fast as possible in order to prune the tree. Breadth first search on the other hand will have as many active nodes as the level before the leaves. Best first search can in worst case be as bad as Breadth first, but generally it is much better. In fact after finding the optimal solution, no unnecessary nodes will be considered, but the growth in memory consumption is still exponential.

In CLP, depth first search is used. When mentioning Branch & Bound and tree search, we implicitly think of *complete search* where all solutions are enumerated and optimum is guaranteed to be found. In some cases a "good" solution is sufficient and perhaps it is not possible to search the entire solution space within reasonable time. Then *incomplete search* can be useful. Here, a number of different strategies are available to heuristically reduce the size of the tree. *Bounded Backtrack Search* simply stops the search after a certain number of backtracks has been exceeded. Other examples are *Credit Search*, *Limited Discrepancy Search* and others described in [1, 3], which more cleverly tries to explore promising regions.

Heuristics: To be able to prune the tree as soon as possible, one needs a reasonably good feasible solution or upper bound before the Branch and Bound is started. Here, a fast construction heuristic can make the difference between solving the problem within seconds and not solving it at all. After finding an initial solution it can be improved by local search heuristics. Local search is well supported by ECLiPSe through the repair library, which allows variables to have tentative values [1, 3]. We will not discuss heuristics further in this paper.

3 Modelling and Solving in ECLiPSe

ECLiPSe can use external LP and MIP solvers as CPLEX and XPRESS-MP and can hence be used purely as a mathematical modeling tool similar to GAMS. We will illustrate this by solving the following *Warehouse Location problem*: Given is a number of customers and a number of warehouse locations. Each open warehouse can only deliver to a limited number of customers and each customer only gets delivery from one warehouse. We must now decide which warehouses to open in order to minimize costs for opening the warehouses and delivering the demanded goods to the customers. The MIP model can be formulated as shown in figure 1 where c_j is the cost of opening warehouse $j \in J$ and d_{ij} is the cost of delivering goods to customer $i \in I$ from warehouse $j \in J$. The binary variable y_j indicates

$$\min \sum_{j \in J} c_j y_j + \sum_{i \in I} \sum_{j \in J} d_{ij} x_{ij} \tag{5}$$

$$\sum_{j \in J} x_{ij} = 1, \qquad i \in I, \tag{6}$$

$$x_{ij} \leq y_j, \quad i \in I,\ j \in J, \tag{7}$$

$$\sum_{i \in I} x_{ij} \leq cap_j, \qquad j \in J, \tag{8}$$

$$y_j \in \{0,1\} \qquad j \in J, \tag{9}$$

$$x_{ij} \in \{0,1\} \quad i \in I,\ j \in J \tag{10}$$

Figure 1: Warehouse Location Model.

the opening of warehouse j while the binary variable x_{ij} indicates delivery to customer i from warehouse j. Constraint (6) guarantees delivery from exactly one warehouse to each customer. (6) forces delivery from only open warehouses. Finally, (6) is the limit of customers serviced by each warehouse.

3.1 Introduction to ECLiPSe

We will for readers not familiar with Prolog or ECLiPSE give a short introduction to constructs used in the following.

- In Prolog the first letter in variables must be a capital letter. If a variable is assigned a value it is said to be *instantiated*. The only way a variable can be assigned another value is by backtracking past the place it was assigned the value and hereby the variable becomes *uninstantiated*.
- A list is indicated by brackets: For `A = [X,Y,Z]`, `A` is instantiated to the list `[X,Y,Z]`.
- `dim(X,[N,M])` can be used for both constructing a matrix `X` or retrieving the dimensions of the matrix. If `X` is a given 2 dimensional matrix, the size is returned in the variables `N` and `M`. On the other hand if `N` and `M` are instantiated and `X` is not, then `X` will be initialized to a matrix with size `N` times `M` consisting of uninstantiated variables.
- `length(X,L)` can be used for both constructing and retrieving the length, `L`, of a list `X` similar to `dim` for matrices.
- `R is X[I,1..M]` sets `R` to the `I`'th row of the matrix `X`. Similarly `C is X[1..N,J]` sets `C` to the `J`'th column of the matrix `X`. `R` and `C` will be lists.
- `Y = 1, X = Y+1` sets the variable `X` equal to the expression `Y+1`.
- `Y = 1, X is Y+1` sets the variable `X` equal to 2. Contrary to "`=`", the expression on the right side evaluated is when using `is`.

The last construct is the loop. We will give several examples to illustrate the different possibilities:

```
(for(I,1,10) do writeln(I))
```

Here the numbers from 1 to 10 are printed by the `writeln` statement.

```
(foreach(X,[1,2,3]) do writeln(X))
```

Prints the numbers 1, 2 and 3 by iterating over the list. `foreach` can also be used for constructing lists:

```
(for(I,1,3), foreach(J,L) do J is I+1), writeln(L)
```

Note that it is not a nested loop, but simply two iterators. Here the list, `L` equals `[2,3,4]` constructed and afterwards `L` is printed outside the loop. Loops can also be used for constructing expressions:

```
(for(I,1,3), foreach(J,L) do J = I+1), writeln(L)
```

Here the list `[1+1,2+1,3+1]` is printed. `I` and `J` are local variables in the loop while `L` is only accessible outside the loop. To make variables accessible within the loop `param` is used:

```
X is 3, (for(I,1,3), foreach(J,L), param(X) do J is I+X), writeln(L)
```

Here `X` instantiated to 3 is available inside the loop and the list `[4,5,6]` is printed outside the loop.

3.2 MIP model in ECLiPSe

The ECLiPSe code of the Warehouse Location model is shown in figures 2 to 5. Calling the function `whouse(Cap,D,C,X,Y,Cost)` will return the optimum value in the variable `Cost`. `Cap` is the vector of capacities, cap_j for warehouses $j \in J$ from the model in figure 1, `D` is the matrix, d_{ij}, with distances from customer i to warehouse j and `C` is the vector of warehouse setup costs, c_j. Line 2 returns the dimensions of the matrix `D`, such that `NCust`=$|I|$ and `NWh`=$|J|$. `NWh` is used for initializing the vector `Y` of y_j's in line 3, and `NCust` and `NWh` are used for initializing the matrix `X` of x_{ij}'s in line 4. In line 5 `cross_prod(1..NCust,1..NWh,Pairs)` returns the list `Pairs` of pairs `[I,J]` for all values of `I` and `J` in the intervals $1, ..., NCust$ and $1, ..., NWh$ to be used later.

```
1: whouse(Cap,D,C,X,Y,Cost) :-
2:        dim(D, [NCust, NWh]),
3:        dim(Y,[NWh]),
4:        dim(X,[NCust,NWh]),
5:        cross_prod(1..NCust,1..NWh,Pairs),
```

Figure 2: Warehouse Location MIP model in ECLiPSe – Data.

In figure 3, the variable domains are defined. Line 6 creates a list `YList` of y_j variables, which are defined to be binary variables in line 7. When solving with an external solver we have indicate that integer variables should be used with the call, `eplex: (integers(YList))`. Otherwise the LP-relaxation is solved instead, which can be useful in other circumstances. In line 9 a list `XList` of x_{ij} variables is extracted from the matrix `X`. In line 10 the variables are again defined to be binary. Here we do not need to to use integer variables, since the constraint matrix is unimodular.

```
 6:        YList is Y[1..NWh],
 7:        YList::0..1,
 8:        eplex: integers(YList),
 9:        matrix_to_list(X,XList),
10:        XList::0..1,
```

Figure 3: Warehouse Location MIP model in ECLiPSe – Variables.

In figure 4 we set up the constraints corresponding to (6), (7) and (8). First we loop over the customers where the variable `I` takes values from 1 to `NCust`. Line 14 constructs a list of x_{ij} variables for row i or here `I`. Line 15 sets up a constraint by summing the variables in the list and requiring this sum to be 1. Lines 16 to 18 iterates over all customer-warehouse pairs setting up constraint (7). Finally for each warehouse j, here `J`, row j is extracted from the matrix in line 21 and constraint (8) is set up in line 22.

```
11:        (for(I,1,NCust),
12:         param(NWh,X)
13:        do
14:         XI is X[I,1..NWh],
15:         eplex: (sum(XI) =:= 1)
           ),

16:        ( foreach([I,J],Pairs),
17:          param(X,Y)
           do
18:          eplex: (X[I,J] =< Y[J])
        ),

19:        ( for(J,1,NWh),
20:          param(X,NCust,Cap)
        do
21:          XJ is X[1..NCust, J],
22:          eplex: (sum(XJ) =< Cap[J])
        ),
```

Figure 4: Warehouse Location MIP model in ECLiPSe – Constraints.

In lines 23 to 26 of figure 5 the customer part of the cost function is collected in the list `CCost` by iterating over all customer-warehouse pairs. Similarly, the warehouse setup costs are collected in lines 27 to 29 and stored in the list `WCosts`. In line 30 the objective function is finally set up and in line 31 solved by calling the external solver with `optimize`.

It is clear that the above program is not as compact as the mathematical formulation or a formulation in another modelling language as for instance GAMS, but it is still relatively short considering it being written in a programming language.

3.3 CLP model in ECLiPSe

Now we set up the same problem as a CLP model shown in figures 6 to 9. Instead of using binary variables, y_j's, we can think of choosing a *set* of warehouses to open. In line 3 we

```
23:        ( foreach([I,J],Pairs),
24:          foreach(CCost,CCosts),
25:          param(D,X)
           do
26:          CCost = X[I,J]*D[I,J]
           ),

27:        ( for(J,1,NWh),
28:          foreach(WCost,WCosts),
             param(C,Y)
           do
29:          WCost = Y[J]*C[J]
           ),

30:        Objective = sum(WCosts)+sum(CCosts),
31:        eplex: optimize(min(Objective),Cost).
```

Figure 5: Warehouse Location MIP model in ECLiPSe – Objective.

initialize a variable, `Y`, with `intset` to be a *set* with numbers between 1 and `NWh`.

Instead of the binary variables, x_{ij}, specifying if customer i is serviced by warehouse j, we introduce variables for each customer taking values corresponding to open warehouses. The variables are initialized in the list `XList` in line 4 and in lines 5-7 they are constrained to take values `in` the set `Y` of open warehouses. We emphasize that `in` is a constraint, which could be mathematically stated like $x_i \in Y$, where x_i is the index of the chosen warehouse of customer i in the set-variable `Y` consisting of indices corresponding to open warehouses.

Note that instead of supplying the warehouse capacities in a vector, we now supply them in a list named `CapList`.

```
 1: whouse(CapList,D,C,XList,Y,Cost) :-
 2:       dim(D, [NCust, NWh]),
 3:       intset(Y, 1, NWh),
 4:       length(XList,NCust),

 5:       ( foreach(X, XList),
 6:         param(Y)
          do
 7:         X in Y
          ),
```

Figure 6: Warehouse Location CLP model in ECLiPSe – Variables.

In lines 8-13 of figure 7 the capacity constraints are set up. We introduce the auxiliary variable `CapVar` in line 12 for each constraint. In line 13 we use the occurrences constraint, which constrains `CapVar` occurrences of warehouse `J` in the list `XList`. In line 14 we add the redundant constraint making the sum of `CapVar`'s equal to the total number of customers. As we will see later redundant constraints can have a significant impact on CPU times.

To get the cost of supplying the customer from the chosen warehouse, we use the constraint `element` in line 20. `CostList` is the list of costs for the given customer and

```
 8:        ( for(J, 1, NWh),
 9:          foreach(CapVar,CapVars),
10:          foreach(Cap,CapList),
11:          param(XList)
           do
12:          fd: (CapVar :: 0..Cap),
13:          occurrences(J,XList,CapVar)
           ),
14:        sum(CapVars) #= NCust,
```

Figure 7: Warehouse Location CLP model in ECLiPSe – Constraints.

`element(X, CostList, CCost)` constrains `CCost` to take the `X`'th element of the list `CostList`. The costs are collected in the list `CCosts` as specified in line 17.

The costs for opening the warehouses, `WCosts`, defined by the set `Y`, is calculated with the `weight` constraint in line 21, where `C` is the vector of setup costs. Basically the weights corresponding to the set `Y` are summed in `WCosts`. For instance if the set is $\{1, 3, 7\}$, then the weights with index 1, 3 and 7 are added together.

```
15:        ( for(I, 1, NCust),
16:          foreach(X, XList),
17:          foreach(CCost, CCosts),
18:          param(Y,D,NWh)
           do
19:          CostList is D[I, 1..NWh],
20:          element(X, CostList, CCost)
           ),
21:        weight(Y,C,WCosts),
22:        Cost #= WCosts + sum(CCosts),
```

Figure 8: Warehouse Location CLP model in ECLiPSe – Objective.

Remaining is now to specify the search procedure in figure 9. In line 24 `minimize` is called. It is a Branch & Bound procedure available in ECLiPSe. The procedure does not use a lower bound, which means that the performance of the search can be very much dependent on, in which order we consider variables and values. Therefore, in line 23 we call a procedure, which for each customer sort the warehouses in increasing distance from the customer to warehouses. In line 26 we specify how to search the set of warehouses to open. `increasing` indicates that the cardinality of the sets we try are increasing, i.e. first we try sets with one element, then two, etc. In line 26 we call the search procedure shown in lines 28 to 32. Here the sorted list of warehouse id's are used to select values of `X`. `member(X, OrderedWarehouseIds)` basically iterates through the different values `X` can take.

3.4 Comparison of CLP and MIP

People from the CLP community would now argue that the CLP formulation is "closer" to how you would think of the real-life problem. OR people would argue instead that MIP models are just as easy to construct with a bit of modelling practice.

```
        order_warehouses(D, CustOrderedWarehouseIds),
        minimize((
            insetdomain(Y, increasing, _, _),
            labeling(XList, CustOrderedWarehouseIds)
        ), Cost).

labeling(XList, CustOrderedWarehouseIds) :-
        ( foreach(X, XList),
          foreach(OrderedWarehouseIds, CustOrderedWarehouseIds)
        do
          member(X, OrderedWarehouseIds)
        ).
```

Figure 9: Warehouse Location CLP model in ECLiPSe – Search.

Another issue comparing CLP and MIP is performance. We have tested the two models on a small example with 19 warehouses and 20 customers for different warehouse capacities. We have also tested different formulations where extra redundant constraints are added in order to improve performance. The following different models are tested:

MIP-S: Formulation in section 3.2.

MIP-LB: The following trivial lower bound on the number of open warehouses is added:

$$\sum_{j \in J} y_j \geq \left\lceil \frac{|I|}{\max_{j \in J} cap_j} \right\rceil \tag{11}$$

MIP-W: It is well known that

$$\sum_{i \in I} x_{ij} \leq cap_j y_j, \quad j \in J \tag{12}$$

is a weaker formulation than (7), but fewer constraints are necessary.

CLP-R: Formulation in section 3.3.

CLP-RLB: CLP-R with addition of same lower bound as for MIP-LB.

CLP-LB: As CLP-RLB without the redundant constraint in figure 7:

`sum(CapVars) #= NCustomers`.

CLP: As CLP-R without the redundant constraint

The tests were done with ECLiPSe 5.5 and XPRESS-MP 13.26 and the results are shown in table 1. All warehouses have the same setup cost and each column reports the results for a given capacity – the same for all warehouses. The necessary number of warehouses are then 2, 3, 4, 5, 7 and 10 for the given capacities. The cost matrix can be supplied by contacting the author.

The results of the aggregated and the stronger version of the MIP formulation are quite surprising. The weaker formulation is much faster on the difficult instances with many warehouses. The explanation is most likely that that the y's are less fractional with many warehouses and the LP problem easier to solve with much less constraints compared to the strong version. Adding the redundant lower bound on the number of warehouses improves the performance significantly.

This is also the case for the CLP formulation. Here the lower bound is the difference making the problem solvable at all. Redundant constraints are often not necessary in MIP models, but for CLP they are. Experiments are as always needed to conclude if the constraints have any effect or if they are useless and even lead to increased runtimes. The other redundant constraint is not as powerful as the lower bound and the combination of the two is worse than the lower bound alone.

The reason the lower bound constraint is so effective, is the way the different sets of open warehouses are tried; first all sets with one warehouse open, then all sets with two open warehouses, etc. For each of the tried sets with too few open warehouses, customers are anyway allocated to warehouses, but without success.

Capacity	10	7	5	4	3	2
MIP-S	0.2	0.3	0.3	0.6	7.7	42.2
MIP-LB	0.2	0.2	0.1	0.1	0.1	0.4
MIP-W	0.2	0.1	0.1	0.1	0.2	0.1
CLP	4.5	318.2	-	-	-	-
CLP-R	4.7	8.8	19.3	50.0	251.3	1081.3
CLP-LB	4.5	7.9	14.8	34.0	126.8	566.8
CLP-RLB	4.7	7.9	15.2	35.0	132.9	585.8

Table 1: Comparison of CLP and MIP for the Warehouse Location Problem.

We have earlier mentioned the importance of trying values in a specific order. In line 23 of figure 9 the warehouses are sorted in order of cost for each customer, which means that when trying to assign warehouses to customers it is done in increasing order of the cost. Without the ordering we get the runtimes shown in table 2, which are significantly higher than CLP-LB.

Capacity	10	7	5	4	3	2
CLP-LB	4.8	8.4	25.14	43.0	158.8	839.3

Table 2: CLP-LB with no ordering

Earlier, we mentioned that branching should be done on important variables first. In this case the most important is, what warehouses to open. To illustrate the impact of making the right choice, we have tried the opposite order with the results shown in table 3.

Capacity	10	7	5	4	3	2
CLP-LB	130.5	456.7	-	-	-	-

Table 3: CLP-LB with branching on customer variables before warehouses.

The basis of the comparison is quite limited – only one quite small and simple prototype example. The intention of the comparison was however not to make the definite choice between MIP and CLP, but to merely illustrate some points.

On this instance of the problem MIP is superior to CLP, but it is not fair to generalize the result to other problems. Problems for which there is a large gap between the upper and lower bounds are not well-suited for MIP. This for instance is the case when disjunctive constraints are necessary in scheduling and packing problems. Here CLP looks particularly promising.

4 Conclusion

We have in this article described CLP from the view point of MIP. The two paradigms have many similarities, but also the potential to complement each other and create even more powerful *hybrid solvers*.

We gave an example of modelling and solving the Warehouse Location Problem in the ECLiPSe environment with both MIP and CLP. We argued that MIP is superior when linear models can be formulated where the gap between upper and lower bounds is relatively small. CLP lacks the ability to use bounding to prune the tree, but can instead use domain reduction to reduce the size of the search tree.

Currently much interesting research is focused on using methods originating from the OR area to do domain reduction in CLP and as well using domain reduction in Branch & Bound. A definite conclusion is not yet near regarding, which components should enter into the future solver of NP-hard problems.

References

[1] A. M. Cheadle, W. Harvey, A.J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace. *ECLiPSe – An Introduction.* IC-Parc, Imperial College, London, August 2002.

[2] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer, 4. edition, 1994.

[3] IC-Parc, Imperial College, London. *ECLiPSe Constraint Library Manual*, 2002.

[4] K. Marriott and P. J. Stuckey. *Programming with Constraints – An Introduction.* MIT Press, 2. edition, 1998.