A NON-BINARY CONSTRAINT ORDERING HEURISTIC FOR CONSTRAINT SATISFACTION PROBLEMS

Miguel A. Salido
Department of Information Systems and Computation
Technical University of Valencia, Spain
msalido@dsic.upv.es

Abstract Nowadays many real problems can be modelled as Constraint Satisfaction Problems (CSPs). A search algorithm for constraint programming requires an order in which variables and values should to be considered. Choosing the right order of variables and values can noticeably improve the efficiency of constraint satisfaction.

Furthermore, the order in which constraints are studied can improve efficiency, particularly in problems with non-binary constraints. In this paper, we present a preprocess heuristic called *Constraint Ordering Heuristic* (COH) that studies the constrainedness of the scheduling problem and mainly classifies the constraints so that the tightest ones are studied first. Thus, constrainedness can be known in advance and overall inconsistencies can be found earlier and the number of constraint checks can significantly be reduced.

Keywords: Heuristic Search, Constraint ordering, Non-binary Constraints, Constrainedness.

1. Introduction

Nowadays, many real problems can be efficiently modelled as Constraint Satisfaction Problems (CSPs) and solved using constraint programming techniques. Some problems can be modelled naturally using non-binary constraints [26, 19, 8]. Although, researchers have traditionally focused on binary constraints [24], the need to address issues regarding non-binary constraints has recently started to be widely recognized in the constraint satisfaction literature. Thus, to define heuristics to solve non-binary CSPs becomes relevant.

One approach to solving CSPs is to use a depth-first backtrack search algorithm [5]. While the worst-case complexity of backtrack search is exponential, several heuristics to reduce its average-case complexity have been proposed in the literature [6]. However, determining which algorithms are superior to others remains difficult. Theoretical analysis provides worst-case guarantees which often do not reflect average performance. For instance, a backtracking-based algorithm that incorporates features such as variable ordering heuristics will often in practice have substantially better performance than a simpler algorithm without this feature [17], and yet the two share the same worst-case complexity. Similarly, one algorithm may be better than another on problems with a certain characteristic, and worse on another category of problem. Ideally, we would be able to identify this characteristic in advance and use it to guide our choice of algorithm.

Many of the heuristics that improve backtracking-based algorithms are based on *variable ordering* and *value ordering* [20, 2], due to the additivity of the variables and values. However, constraints are also considered to be *additive*, that is, the order of imposition of constraints does not matter, all that matters is that the conjunction of constraints be satisfied [2].

Thus, a real problem can be modelled as a CSP, and using some of the current ordering heuristics, it can be solved in a more efficient way by some of the backtracking-based search algorithms (Figure 1).

In spite of the additivity of constraints, only a few works have be done on binary constraint ordering mainly for arc-consistency algorithms [25, 16, 10], but little work has be done on non-binary constraint ordering (for instance in disjunctive constraints [22]), and only some heuristic techniques classify the non-binary constraints by means of the arity. However, less arity does not imply a tighter constraint. Moreover, when all non-binary constraints have the same arity, or these constraints are classified as hard and soft constraints, these techniques are not useful.

In this paper, we propose a heuristic technique called *Constraint Ordering Heuristic* (COH) that can easily be applied to any backtracking-based search algorithm. COH studies the constrainedness of the problem and mainly classifies the constraints so that the tightest ones are studied first. This is based on the *first-fail* principle, which can be explained as: *"To succeed, try first where you are more likely to fail".*

In this way, the tightest constraints are selected first for constraint checking. Thus, the search space is pruned soon and inconsistent tuples can be found earlier because it is not necessary to check these inconsistent tuples with the rest of the constraints with the corresponding savings in constraint checking.

In the following section, we formally define constraint satisfaction problems and describe two well-known ordering heuristic method/techniques. In section 3, we describe the Constraint Ordering Heuristic. In section 4, we present the computational complexity. An application of COH to Railway Scheduling Problem is presented in section 5. Section 6 presents the results of the evaluation. Finally, in section 7, we present our conclusions.

2. Definition and Algorithms

Briefly, a constraint satisfaction problem (CSP) consists of:

- a set of variables $X = \{x_1, x_2, ..., x_n\}$
- each variable $x_i \in X$ has a set D_i of possible values (its domain)
- a finite collection of constraints $C = \{c_1, c_2, ..., c_k\}$ restricting the values that the variables can simultaneously take.

A solution to a CSP is an assignment of values to all the variables so that all constraints are satisfied; a problem with a solution is termed *satisfiable* or *consistent*. The objective in a CSP may be to determine:

- whether a solution exists, that is, if the CSP is consistent
- all solutions, many solutions, or only one solution, with no preference as to which one
- an optimal, or a good solution by means of an objective function defined in terms of certain variables.

In some problems it is desirable to find all solutions in order to give the user the ability to search the design space for the best solution, particularly when various parameters are difficult to model [9]. Some techniques such as value ordering are not valid to solve this type of problems.

Two ordering heuristics for CSPs are analysed in [20, 2]: variable ordering and value ordering. Let's briefly look at these two heuristics.

2.1 Variable Ordering

The experiments and analyses by several researchers have shown that the ordering in which variables are assigned during the search may have substantial impact on the complexity of the search space explored. The ordering may be either a static ordering, or dynamic ordering. Examples of static ordering heuristics are *minimum width* [12] and *maximum degree* [7], in which the order of the variables is specified before the search begins, and it is not changed thereafter. An example of dynamic ordering heuristic is *minimum remaining values* [17], in which the choice of next variable to be considered depends on the current state of the search.

Dynamic ordering is not very useful for all search algorithms, e.g., with simple backtracking, during the search, there is no many extra information available that could be used to make a different choice of ordering from the initial ordering. However, with forward checking, the current state includes the domains of the variables as they have been pruned by the current set of instantiations. Therefore, it is possible to base the choice of the next variable on this information.

2.2 Value Ordering

Comparatively little work has been done on algorithms for value ordering even for binary CSPs [14, 13]. The basic idea behind value ordering algorithms is to select the value for the current variable which is most likely to lead to a solution. Again, the order in which these values are considered can have substantial impact on the time necessary to find the first solution. However, if all solutions are required or the problem is not consistent, then the value ordering does not make any difference. A different value ordering will rearrange the branches emanating from each node of the search tree. This is an advantage if it ensures that a branch which leads to a solution is searched earlier than a branch which leads to a dead-end. For example, if the CSP has a solution, and if a correct value is chosen for each variable, then a solution can be found without any backtracking.

Suppose we have selected a variable to instantiate: how should we choose which value to try first? It may be that none of the values will succeed. In that case, every value for the current variable will eventually have to be considered and the order does not matter. On the other hand, if we can find a complete solution based on the past instantiations, we

want to choose a value which is likely to succeed and unlikely to lead to a conflict.

3. Constraint Ordering Heuristic (COH)

Currently, several constraint satisfaction techniques have been developed to directly manage non-binary CSPs due to transforming techniques have many drawbacks. Some main drawbacks are:

- Transforming a non-binary into a binary CSP produces a significant increase in the problem size, so the transformation may not be practical [3, 18] particulary in large real problems.
- A forced binarization generates unnatural formulations, which cause extra difficulties for CSP solver interfaces with human users [4].

As current techniques manage non-binary constraints in a natural way [4], new heuristics can be applied to these constraints to reduce the search space. As we pointed out in introduction some of these heuristics classify the non-binary constraints by means of the arity. However, when all non-binary constraints have the same arity (maximum arity), these techniques are not useful.

In this section, we present a heuristic technique called *Constraint Ordering Heuristic* (COH) for two different purposes:

1 to study the constrainedness of the problem. We introduce a parameter (τ) that measures the constrainedness of a problem. This parameter represents the probability of a problem being feasible. A value of $\tau = 0$ corresponds to an over-constrained problem and no states are expected to be solutions. A value of $\tau = 1$ corresponds to an under-constrained problem and every state is a solution.

2 to classify the non-binary constraints, independently of the arity so that the tightest constraints are studied first. Inconsistencies can then be found earlier and the number of constraint checks can be significantly reduced.

3.1 Specification of COH

COH is a preprocess heuristic based on the sampling from a finite population, as in statistics, where there is a population, and a sample is chosen to represent this population. In our context, the population is composed of the points (states) lying within the convex hull of all initial stated generated by means of the Cartesian product of variable domain bounds. The sample is composed by $s(n, d, e)$ points distributed

by a simple random or systematic sampling where s is a function that depends on the number of variables (n), domain size (d) and level of precision (e).

As in statistic, the user selects the desired precision by the size of the sample $s(n, d, e)$. Generally, as larger the size is, better the classification is. Yamade in [27] provides a simplified formula to calculate sample sizes. This formula in our context can be viewed as:

$$s(n, d, e) := \lceil \frac{d^n}{1 + d^n \cdot e^2} \rceil \tag{1}$$

where n is the number of variables, d is the maximum domain size and e is the level of precision. When this formula is applied to the four-queen problem (see section 3.1.2), with a 75% confidence level, we get Equation (2).

$$s(4, 4, 0.25) := \lceil \frac{256}{1 + 256 \cdot 0.25^2} \rceil = 16 \tag{2}$$

Figure 2 represents the graphic of the simplified formula for proportion presented by Yamane. This curve has a logarithmic appearance by independence of the population size and the level of precision. Figure 2 shows the curve where $4^4 = 256$ is the population size and 75% is the confidence level. The sample size is 16 as we will see in section 3.1.2.

Depending on the desired level of precision, the sample size will be bigger of shorter, but with acceptable level of precision (between 75% and 85%) the sample size is polinomial with the number of variables and domain size ($n \cdot d$ and $\frac{1}{2} n \cdot d^2$ respectively).

After the sample is taken, for each constraint c_i each point in the sample is projected over the variables contained in c_i and then the consistency of the resulting tuple is checked. Thus, each constraint c_i is labeled with p_i: $c_i(p_i)$, where $p_i = st_i/s(n, d, e))$ represents the proportion of possible states, that is, the tightness of the constraint.

We present the pseudo-code of COH below.

Constraint Ordering Heuristic

Inputs: A set of n variables, $X_1, ..., X_n$;
For each X_i, a set D_i of possible values (the domain)
A set of constraints, $C_1, ..., C_k$.

Outputs: A set of labels corresponding with the tightness of each constraint, $\{p_1, ..., p_k\}$.

1.- From the entire number of points generated by the Cartesian Product of the variable domain bounds, COH selects a well distributed sample with $s(n)$ points.

2.- With the selected sample of points ($s(n, d, e)$), COH studies how many points $st_i : st_i \leq s(n, d, e)$ satisfy each constraint c_i. c_i is labelled with p_i such that $p_i = st_i/s(n, d, e)$.

3.- COH obtains a set of labels that represents the tightness of each constraint, $\{p_1, ..., p_k\}$

Once each constraint is labelled with its tightness, COH uses this information for two different purposes abovementioned:

- Given $\{p_1, ..., p_k\}$, we introduce a new parameter called τ that measures the constrainedness of the problem. This parameter represents the probability of a problem being feasible. This parameter lies in the range $[0, 1]$. A value of $\tau = 0$ corresponds to an over-constrained and no state is expected to be a solution ($\langle Sol \rangle = 0$). A value of $\tau = 1$ corresponds to an under-constrained and every state is expected to be a solution ($\langle Sol \rangle = \prod_{v \in V} d_v$).

  Thus, given the set of probabilities $\{p_1, ..., p_k\}$, the number of solutions can be computed as:

$$\langle Sol \rangle := (\prod_{v \in V} d_v) \times (\prod_{c_i \in C} (p_{c_i})) \tag{3}$$

  This equation is equivalent to the obtained in [15]. However, our definition of constrainedness is given by the following equation:

$$\tau := \prod_{c_i \in C} (p_{c_i}) \tag{4}$$

  τ is a parameter that measures the probability that a randomly selected state is a solution, that is, the probability this selected state satisfies the first constraint (p_1), the second constraint (p_2)

and so forth, the probability this state satisfies the last constraint (p_k). Thus, this parameter lies in the range $[0, 1]$ that represents the constrainedness of the problem.

Once, we know the constrainedness of the problem, we can solve the problem with an appropriate CSP solver. For instance, if the problem is under-constrained, an easy heuristic can be applied for solving it. However, if the problem is hard to solve, a complete solver must be applied for solving it.

- Given $\{p_1, ..., p_k\}$, COH classifies the constraints in ascending order of the labels p_i so that the tightest constraints are classified first $(C_{ord1}, C_{ord2}, ..., C_{ordk})$. Therefore, COH translates the initial non-binary problem into an ordered non-binary problem so that it can be studied by a CSP solver (see Figure 3).

3.1.1 Example 1:(Continuous Problem). Let's assume a continuous problem with two variables ($n = 2$) $x_1, x_2 : [0, 4]$, and three constraints ($k = 3$): $c_1 : 3x_1 + 2x_2 \leq 14$, $c_2 : 2x_1 - 0.25x_2 \leq 5$, and $c_3 : x_1 - x_2 \leq -3.75$ (see Figure 4).

COH checks how many points (from a given sample: $s(n, d, e) = n^2 = 4$ points) satisfy each constraint and classifies them afterwards. The selected points are (0,0), (0,4),(4,0),(4,4) and the following results are obtained:

$$st_1 = 3 \quad st_2 = 2, \quad st_3 = 1$$
$$p_1 = 3/4 = 0.75, \quad p_2 = 2/4 = 0.50, \quad p_3 = 1/4 = 0.25$$

$$\begin{array}{ccc} c_1 : (0.75) & & c_3 : (0.25) \\ c_2 : (0.50) & \overrightarrow{ordering} & c_2 : (0.50) \\ c_3 : (0.25) & & c_1 : (0.75) \end{array}$$

Figure 4, shows the behaviour of our CSP solver *Hyperpolyhedron Search Algorithm* (HSA) [21]. HSA is a CSP solver that manages non-binary constraints by means of Polytopes, in which faces represent the problem constraints and vertices represent the extreme solutions. If HSA carries out the consistency study in the order of imposition of constraints (option 1) (c_1, c_2, c_3), HSA will generate 6 new vertices. However, if HSA runs the constraint ordering heuristic, which classifies the constraints in ascending order (option 2) (c_3, c_2, c_1), HSA will generate only two new vertices with the corresponding time reduction.

Table 1 shows the number of constraint checks in the same problem executed using Backtracking (BT) and Backtracking with COH (BT+COH). BT checks all constraints in each non-valid instantiation while BT+COH only checks the tightest one (c_3). Thus, to study the

problem consistency (one solution), BT has made 15 constraint checks while BT+COH only 7 constraint checks. However, if all solutions are required, BT will make 75 constraint checks and BT+COH will make only 27 constraint checks.

3.1.2 Example 2: (The 4-Queens Problem). This well-known problem is an example of discrete problem with four variables, and seven constraints.

Figure 5 shows the initial CSP and the possible solutions (2,4,1,3) and (3,1,4,2) obtained using BT and BT+COH. COH checks how many tuples (from a given sample: $s(n,d,e) = 16$ tuples (see equation (2)) $\{(1,1),(1,2),\cdots,(4,3),(4,4)\}$) satisfy each constraint and classifies them afterwards. We can observe that some constraints are tighter than others (c_1, c_4, c_6 satisfy less valid tuples). BT must check 224 constraints to obtain both solutions, while BT+COH must only check 168. It can be observed that BT+COH studies first the tightest constraints: c_1, c_4, c_6. These constraints are enough to obtain both solutions and the following only must be checked to be consistent.

$$(2,4,1,3) \leftarrow c_1:(2,4), c_4:(4,1), c_6:(1,3)$$
$$(3,1,4,2) \leftarrow c_1:(3,1), c_4:(1,4), c_6:(4,2)$$

In section 6, we present a more exhaustive evaluation of the n-queens problem.

4. Analysis of COH

COH selects a sample composed of $s(n,d,e)$ points, so the spatial cost is $O(s(n,d,e))$. COH checks the consistency of the sample with each non-binary constraint, so its temporal cost is $O(ks(n,d,e))$. Then, COH classifies the set of constraints in ascending order. Its temporal complexity is $O(klogk)$. Thus, the temporal complexity of COH is $O(max\{ks(n,d,e), klogk\})$.

5. Application of COH to Railway Scheduling Problem

Train scheduling has been a significant issue in the railway industry. Numerous approaches and tools have been developed to compute railway scheduling. We summarize the application of COH into our constraint-based train scheduling tool[1] [1], which is a project in collaboration with the National Network of Spanish Railways (RENFE). We formulate train scheduling as constraint optimization problems. COH has been inserted into our heuristics to speed up and direct the search towards suboptimal

solutions in periodic train scheduling problems. The feasibility of our problem-oriented heuristics was confirmed with experimentation using real-life data. The results show that our techniques enable MIP solvers such as LINGO and ILOG Concert Technology (CPLEX©) to terminate earlier with good solutions.

The railway scheduling problem can be described as a constraint optimization problem, where the main objective function is to minimize the journey time of all trains. Variables are frequencies, arrival and departure times of trains at stations and binary auxiliary variables generated for modelling disjunctive constraints. Constraints are composed by user requirements, traffic rules, and topological constraints. These constraints are composed by the parameters defined by user interfaces and database accesses.

The formal mathematical model is presented in Model 1. Let's suppose a railway network with r stations, n trains running in the down direction, and m trains running in the up direction. We assume that two connected stations have only one line connecting them. T_iA_k represents that train i arrives at station k; T_iD_k means that train i departs from station k; $Timei_{k-(k+1)}$ is the journey time of train i to travel from station k to $k+1$; TSi_k and CSi_k represent the technical and commercial stop times of train i in station k, respectively; and ET_i and RT_i are the expedition and reception time of train i, respectively.

The main complexity of the problem derives in solving the MIP problem due to the binary (integer) variables. If we are able to assign values to these integer variables, the linearized problem can be solved more efficiently. COH carries out a constraint partition in two different sets of constraints: constraints with integer variables and constraints with continuous variables. In both sets of constraints, COH studies the constrainedness of the subproblems and classifies the constraints in the appropriate order. The first set of constraints are composed by constraints where integer variables are involved. These constraints correspond to 'crossing constrains' (6.1)(6.2), 'expedition time constrains' (7.1)(7.2) and 'reception time constrains' (8.1)(8.2). These constraints (ordered by COH) are solved by our heuristics by means of topological techniques. The constrainedness of the subproblem given by COH is useful for our heuristics to direct the search.

In the next section, we present the advantage of inserting COH in our heuristics.

(1) **Min** $\sum_{i=1}^{i=n}(T_iA_r - T_iD_1) + \sum_{j=1}^{i=m}(T_jA_1 - T_jD_r)$;
Subject To
/frequency constraint $\forall i = 1..n, \forall k = 1..r$
(2) $T_{i+1}D_k - T_iD_k = Frequency$;
/Time Constrains $\forall i = 1..n, \forall k = 1..r$
(3.1) $T_iA_{k+1} - T_iD_k = Timei_{k-(k+1)}$;
(3.2) $T_jA_k - T_iD_{k+1} = Timei_{k-(k+1)}$;
/Stations Time Constrains $\forall i = 1..n, \forall k = 1..r$
(4) $T_iD_k - T_iA_k - TSi_k = CSi_k$;
/Constrains to limit journey time $\forall i = 1..n, \forall j = 1..m$
(5.1) $T_iA_r - T_iD_1 \leq (1 + \frac{\delta}{100}) * Timei_{1-r}$;
(5.2) $T_jA_1 - T_jD_r \leq (1 + \frac{\delta}{100}) * Timej_{r-1}$;
/Crossing Constrains $\forall i = 1..n, \forall j = 1..m, \forall k = 1..r$
(6.1) $T_jA_k - T_iD_k <= 86400 * Y_{i-j;k-(k+1)}$;
(6.2) $T_iA_{k+1} - T_jD_{k+1} <= 86400 * (1 - Y_{i-j;k-(k+1)})$;
/Expedition time constrains $\forall i = 1..n, \forall j = 1..m, \forall k = 1..r$
(7.1) $T_jA_k - T_iD_k - 86400 * (X_{i-j} - Y_{i-j;k-(k+1)} + Y_{i-j;(k+1)-(k+2)} - 1) + ET_i <= 0$;
(7.2) $T_iA_k - T_jD_k - 86400 * (X_{i-j} - Y_{i-j;k-(k+1)} + Y_{i-j;(k+1)-(k+2)} - 2) + ET_j <= 0$;
/Reception time constrains $\forall i = 1..n, \forall j = 1..m, \forall k = 1..r$
(8.1) $T_iA_k - T_jA_k - 86400 * (X_{i-j} - Y_{i-j;k-(k+1)} + Y_{i-j;(k+1)-(k+2)} - 1) + RT_i <= 0$;
(8.2) $T_jA_k - T_iA_k - 86400 * (X_{i-j} - Y_{i-j;k-(k+1)} + Y_{i-j;(k+1)-(k+2)} - 2) + RT_j <= 0$;
/Binary Constraints
X_{i-j}; $\forall i = 1..n, \forall j = 1..m$
$Y_{i-j;k-(k+1)}$; $\forall i = 1..n, \forall j = 1..m, \forall k = 1..r$

Model 1: Formal Mathematical Model of the Railway Scheduling Problem.

6. Evaluation of COH

In this section, we compare the performance of COH with some well-known CSP solvers: Chronological Backtracking (BT), Generate&Test (GT), Forward Checking (FC) and Real Full Look Ahead (RFLA) implemented in a tool called CON'FLEX [2], because they are the most appropriate techniques for observing the number of constraint checks.

As we have pointed out in introduction determining which algorithms are superior to others remains difficult. Algorithms and heuristics have often been compared by observing their performance on benchmark problems, such as the n-queens puzzle, or on suites of random instances generated from a simple, uniform distribution. The advantage of using a benchmark problem is that if it is an interesting problem (to someone), then information about which algorithm works well on it is also interesting. The drawback is that if an algorithm beats to any other algorithm on a single benchmark problem, it is hard to extrapolate from this fact. An advantage of using random problems is that there are many of them, and researchers can design carefully controlled experiments and report

averages and other statistics. A drawback of random problems is that they may not reflect real life situations.

This empirical evaluation was carried out with both different types of problems: benchmark problems and random problems. Here, we mainly analyzed the number of constraints checks as a measure of efficiency. In many cases, the number of variables, the domain size and the number of constraints is low to analyze the number of constraint check or the number of constraint check saving.

Benchmark problems

The n-queens problem is a classical search problem to analyse the behaviour of algorithms. In previous section, the 4-queens problem was internally studied.

The 4-queens problem is the simplest instance of the n-queens problem with solutions. The problem is to place four queens on a 4×4 chessboard so that no two queens can capture each other. That is, no two queens are allowed to be placed on the same row, the same column, or the same diagonal. In the general n-queens problem, a set of n queens is to be placed on an $n \times n$ chessboard so that no two queens attack each other.

In Table 2, we present the amount of constraint check saving in the n-queens problem using Generate and Test with COH (GT+COH), Backtracking with COH (BT+COH), Forward Checking with COH (FC+COH) and Real Full Look Ahead with COH (RFLA+COH). Here, our objective is to find all solutions. The results are proportionally extended up to 200 queens. The results show that the amount of constraint check saving was significant in GT+COH and BT+COH due to the fact that COH classified the constraints in the appropriate order, so that the tightest constraints were checked first, and inconsistent tuples were discarded earlier. Furthermore, the amount of constraint check saving was also significant in FC+COH and RFLA+COH in spite of being more powerful algorithms than BT and GT.

France and Thornley, in [11], present a benchmark about a continuous optimization problem (see Figure 6). BT+COH needed 199 constraints checks to solve the problem, while BT needed 340 constraints checks.

In this section, we also compare the performance of COH inserted in one of our heuristics for filtering railway optimization problems. To this end, the size of the sample was $s(n, d, e) = n^2$, where n was the number of trains in each direction. These problems were finally solved using well-known tools such as CPLEX and LINGO.

This empirical evaluation was carried out on a real railway infrastructure that joins two important Spanish cities ("*La Coruña*" and "*Vigo*").

The journey between these two cities is currently divided by 40 stops between stations (23) and halts (17).

In this empirical evaluation, each set of instances was defined by the 3-tuple $< n, s, f >$, where n was the number of trains in each direction, s the number of stations/halts and f the frequency. The problems were generated by modifying these parameters. Thus, each part of the table shown sets two of the parameters and varies the other one in order to evaluate the algorithm performance when this parameter increases. It must be taken into account that runtime of the form "$> xh$." represents that the problem did not finish in x hours and the best solution found up to date is presented in the journey time column.

Heuristic 1 is also called *complete*. This heuristic carries out a filtering over the set of constraints from the formal mathematical model presented in Model 1. Many constraints of type (6) (7) and (8) can be removed according to their departure times and maximum slacks. If a train going in the down direction arrives at the destination before a train going in the up direction departs, then both trains will not cross each other. Thus, a huge number of constraints and integer variables we can eliminated.

Heuristic 2 is a metaheuristic based on heuristic 1. This heuristic carries out a search over the binary variables. Once many integer variables have been removed by heuristic 1, a new filtering process on the reduced problem can eliminate other integer variables by means of local search. Instead of assigning a random station as a crossing station between two opposite trains, heuristic 2 performs a linearized execution where the integer variables have been transformed into continues ones. Thus, the crossing between two trains may not be assigned in stations but on a track between two stations. This will be the initial point to start the search to find the station where the crossing will finally be performed.

Heuristic 2+COH is a similar version of Heuristic 2. Heuristic 2+COH separates the original formulation (MIP problem) into two different subproblems: First, the crossings are solved and then the running maps are calculated. Thus, the problem constraints are classified so that most restricted constraints are studied first. Furthermore, the study of crossings will be partitioned into a set of subproblems so that the solution of each subproblem will generate a traffic pattern. The partition is carried out through the stations that take part in the running map. Each block of the partition is composed by contiguous stations so that each traffic pattern represents the running map corresponding to each block of stations.

In Table 3 (a), we present the runtime and the journey time in problems where the number of trains was increased from 5 to 75, and the number of stations/halts and the frequency were set at 40 and 90, respec-

tively: $< n, 40, 90 >$. The results show as the number of trains increased, the runtime of heuristic 1 and 2 (without COH) was worse. Heuristic 1 obtained the optimal solution for 5,10,15 and 20 trains. However for 50 and 75 trains, heuristic 1 was aborted in 5 hours and the best solutions are similar than obtained by heuristic 2+COH. However heuristic 2+COH had a lower runtime, due to COH classified the tightest constraints first (with integer variables). This heuristic is independent of the number of trains due to symmetry. Figure 7 shows the system interface executing our heuristic 2+COH with the instance $< 10, 40, 90 >$. The first window shows the user parameters, the second window presents the best solution obtained at that point, the third window presents data about the best solution found, and finally the last window shows the obtained running map.

Table 3 (b) shows the runtime and the journey time in problems where the number of stations was increased from 10 to 60, and the number of trains and the frequency were set at 10 and 90, respectively: $< 10, s, 90 >$. In this case, only stations were included to analyze the behavior of the techniques. It can be observed that heuristic 2+COH was better than the others obtaining optimal solutions for 10,20 and 40 stations and better solutions for 30 and 60 stations. Up to 30 stations heuristic 2 had better behaviour than heuristic 1 (complete heuristic). It is important to note the difference between the instance $< 10, 40, 90 >$ of the Table 3 (a) and the instance $< 10, 40, 90 >$ in Table 3 (b). They represent the same instance; however in Table 3 (b) we only used stations (no halts, where crossing is not possible), so the number of possible crossing between trains was much larger. This item reduced the journey time from 2:22:08 to 2:20:22, but the number of combinations increased the running time from 3" to 6".

In Table 3 (c), we present the runtime and the journey time in problems where the frequency was decreased from 140 to 60 and the number of trains and the number of stations were set at 20 and 40, respectively: $< 20, 40, f >$. As the frequency decreased, the process solving become harder. The quality of the solutions depends mainly of the network topology. For this reason, heuristic 2+COH obtained better journey times with frequencies of 100 and 75 minutes, and worse journey time with other frequencies but in lower runtime.

In general, Heuristic 2+COH had good journey times in a reduced runtime over many instances of Table 3. This is, due to, COH has identified the bottlenecks of the problem by the constrainedness of the problem. Once, the bottlenecks have been solved, the problem is easily solved.

Random problems

As we pointed out, benchmark sets are used to test algorithms for specific problems, but in recent years, there has been a growing interest in the study of the relation among the parameters that define an instance of CSP in general (i.e., the number of variables, domain size and arity of constraints). Therefore, the notion of randomly generated CSPs has been introduced to describe the classes of CSPs. These classes are then studied using empirical methods.

In this case, each set of random constraint satisfaction problems was defined by the 3-tuple $< n, c, d >$, where n was the number of variables, c the number of constraints and d the domain size. The problems were randomly generated by modifying these parameters. We considered all constraints as global constraints, that is, all constraints had maximum arity in order to analyze deeper the amount of constraint checks. As in the previous evaluation, each of the tables shown sets two of the parameters and varies the other one in order to evaluate the algorithm performance when this parameter increases. We evaluated 100 test cases for each type of problem and each value of the variable parameter.

Table 4 presents an evaluation of our parameter τ that measures the constrainedness of the random instance. Thus, we can evaluate our estimator with the real one in order to obtain the percentage of error. Thus, Table 4 presents the average real constrainedness by obtaining all solutions, our estimator τ choosing a sample of $s(n, d, e) = 7n^2$ states, the number of possible states, the average number of possible solutions, the average number of estimate solutions using τ and the error percentage. For example in problems with 5 variables, each with 5 possible values and 5 constraints $< 5, 5, 5 >$, the number of possible states is $d^n = 5^5 = 3125$, the average number of solutions is 125, so the actual constrainedness is 0.04. With a sample of $7n^2 = 175$ states, we obtain an average number of 6.64 solutions. Thus, our parameter $\tau = 0.038$ and the number of estimate solutions of the entire problem is 118.7. In this way, the error percentage is only 0.2%.

Following, in tables 5, 6, 7 and 8, we study the constraints checking by selecting a sample of $s(n, d, e) = n \cdot d$ random states.

In Table 5, we present the number of constraint checks in problems solved with BT and BT+COH, where the number of constraints was increased from 3 to 15 and the number of variables and the domain size were set at 3 and 20, respectively: $< 3, c, 20 >$. The results show that the number of constraint checks was significantly reduced in BT+COH in consistent problems (S) as well as in non-consistent problems (F). It must be taken into account that BT must check 138915 constraints in

non-consistent (F) problems with 15 constraints, while BT+COH must only check 9761, due to the tightest constraints remained the problem non-consistent.

In Table 6, we present the number of constraint checks in problems where the number of constraints was increased from 3 to 15 and the number of variables and the domain size were set at 5 and 10, respectively: $< 5, c, 10 >$. In this case the *arc-consistency* filtering algorithm was applied before BT and BT+COH in order to observe the behaviour of COH in previously filtered problems. Therefore, this table presents the evaluation of BT-AC and BT-AC+COH. In this case, we only present the consistent problems, because most of the non-consistent problems were detected by the arc-consistency algorithm. The results show that the number of constraint checks were reduced in all cases.

In Table 7, we present the number of constraint checks in problems, solved with BT and BT+COH, where the domain size was increased from 5 to 35 and the number of variables and the number of constraints were set at 3 and 5, respectively: $< 3, 5, d >$. As the size of the domain increased, the number of constraint checks also increased in consistent problems (S) as well as in non-consistent problems (F).

Finally, in Table 8, we present the evaluation of BT-AC and BT-AC+COH in problems where the domain size was increased from 5 to 35 and the number of variables and the number of constraints were set at 3 and 5 respectively: $< 3, 5, d >$. As in the above tables, the number of constraint checks was reduced when COH classified the constraints.

7. Conclusion and future work

In this paper, we present a heuristic technique called *Constraint Ordering Heuristic* (COH) that can be applied to any backtracking-based search algorithm to solve real problems. This heuristic studies the constrainedness of the problem and mainly classifies the constraints so that the tightest ones are studied first. Thus, inconsistent tuples can be found earlier with the corresponding savings in constraint checking. Also, hard problems can be solved more efficiently overall in problems where many (or all) solutions are required. Furthermore, this heuristic technique has also been modelled by a multi-agent system [23] in which agents are committed to solve their own subproblems.

For future work, we are working on a combination of COH with a variable ordering heuristic in order to manage efficiently more complex problems.

Acknowledgments

This work has been partially supported by the research projects TIN2004-06354-C02-01 (Min. de Educacion y Ciencia, Spain-FEDER), FOM-70022/T05 (Min. de Fomento, Spain), GV/2007/274 (Generalidad Valenciana) and by the Future and Emerging Technologies Unit of EC (IST priority - 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

Notes

1. More information can be found in http://www.dsic.upv.es/users/ia/gps/MOM

2. Backtracking, Generate and Test, Forward Checking and Real Full Look Ahead were obtained from CON'FLEX, which is a C++ solver that can handle non-binary constraint with discrete and continuous domains. It can be found in: http://www-bia.inra.fr/T/conflex/Logiciels/adressesConflex.html.

References

[1] Barber, F., Salido, M.A., Abril, M., Ingolotti, L., Lova, A., Tormos, P.: 2004, 'An Interactive Train Scheduling Tool for Solving and Plotting Running Maps'. *Current Topic in Artificial Intelligence* LNCS 3040, pp. 646–655.

[2] Barták, R.: 1999, 'Constraint Programming: In Pursuit of the Holy Grail'. *in Proceedings of WDS99 (invited lecture), Prague, June.*

[3] Bessire, C.: 1999, 'Non-Binary Constraints'. *In Proc. Principles and Practice of Constraint Programming (CP-99)* pp. 24–27.

[4] Bessire, C., Meseguer, P., Freuder, E. and Larrosa J.: 1999, 'On Forward Checking for Non-binary Constraint Satisfaction'. *Artifical Intelligence* pp. 205–224.

[5] Bitner, J. and E. Reingold: 1975, 'Backtracking Programming Techniques'. *Communications of the ACM 18* pp. 651–655.

[6] Dechter, R. and J. Pearl: 1988, 'Network-Based Heuristics for Constraint Satisfaction Problems'. *Artificial Intelligence* 34, pp. 1–38.

[7] Dechter, R. and I. Meiri: 1994, 'Experimental Evaluation of Preprocessing Algorithms for Constraints Satisfaction Problems'. *Artificial Intelligence* 68, pp. 211–241.

[8] Frost, D. and Dechter: 1999, 'Maintenance scheduling problems as benchmarks for constraint algorithms'. *Annals of Mathematics and Artificial Intelligence* 26, pp. 149–170.

[9] Denk, T. and Parhi, K.: 1998, 'Exhaustive Scheduling and Retiming of Digital Signal Processing Systems'. *in IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing* 45, pp. 821–837.

[10] Van Dongen, M.R.C.: 2002, 'AC-3d, an Efficient Arc-Consistency Algorithm with a Low Space Complexity'. *Proc. of the Eigth International Conference on Principles and Practice of Constraint Programming* LNCS 2470, pp. 755 - 760.

[11] France, J. and Thornley, J.: 1984, *Mathematical models in Agriculture.* Butterworths.

[12] Freuder, E.: 1982, 'A Sufficient Condition for Backtrack-Free Search'. *Journal of the ACM* 29, pp. 24–32.

[13] Frost, D. and Dechter, R.: 1995, 'Look-ahead value orderings for constraint satisfaction problems'. *In Proc. of IJCAI-95* pp. 572–578.

[14] Geelen, P.: 1992, 'Dual viewpoint heuristic for binary constraint satisfaction problems'. *In proceeding of European Conference of Artificial Intelligence (ECAI'92)* pp. 31–35.

[15] Gent, I.P., MacIntyre, E., Prosser, P. and Walsh, T.: 1996 'The Constrainedness of Search', *In Proceedings of AAAI-96*, pp. 246–252.

[16] Gent, I.P., MacIntyre, E., Prosser, P. and Walsh, T.: 1997 'The constrainedness of arc consistency', *Principles and Practice of Constraint Programming*, pp. 327–340.

[17] Haralick, R. and Elliot G.: 1980, 'Increasing Tree Efficiency for Constraint Satisfaction Problems'. *Artificial Intelligence* 14, pp. 263–314.

[18] Larrosa, J.: 1998, *Algorithms and Heuristics for total and partial Constraint Satisfaction.* Barcelona: Phd Dissertation, UPC.

[19] Ros, L., Creemers, T., Tourouta, E. and Riera, J.: 2001, 'A Global Constraint Model for Integrated Routeing and Scheduling on a Transmission Network'. *in Proc. 7th International Conference on Information Networks, System and Technologies*, Minsk, Belarus.

[20] Sadeh, N. and Fox, M.: 1995, 'Variable and Value Ordering Heuristics for the Job Shop Scheduling Constraint Satisfaction Problem'. *tech. report CMU-RI-TR-95-39, Robotics Institute, Carnegie Mellon University.*

[21] Salido, M.A. and Barber, F.: 2001, 'An Incremental and Non-binary CSP Solver: The Hyperpolyhedron Search Algorithm'. *In Proc. of 7th International Conference on Principles and Practice of Constraint Programming (CP-01), LNCS 2239* pp. 779–780.

[22] Salido M.A. and Barber, F.: 2003 'A polynomial algorithm for continuous non-binary disjunctive CSPs: extended DLRs', *Knowledge-Based Systems*, 16, pp. 277–285.

[23] Salido M.A. and Barber, F.: 2006 'Distributed CSPs by Graph Partitioning', *Applied Mathematics and Computation* 183, pp. 491–498.

[24] Tsang, E.: 1993, *Foundation of Constraint Satisfaction.* London and San Diego: Academic Press.

[25] Wallace, R. and Freuder, E.: 1992 'Ordering heuristics for arc consistency algorithms', *In Proc. of Ninth Canad. Conf. on A.I.*, pp. 163–169.

[26] Wallace, M.: 1994, 'Applying constraints for scheduling'. *In Constraint Programming* p. 131 of NATO ASI Series Advanced Science Institute Series.

[27] Yamade, T.: 1967 'Statistics, An Introductory Analysis', *2nd Ed., New York: Harper and Row.*.

Figure Captions:

Figure 1: Constraint Ordering

Figure 2: The Yamane's simplified formula for proportions in the 4-queens problem.

Figure 3: From non-ordered constraint to ordered constraint.

Figure 4: Constraint Ordering Heuristic using HSA.

Figure 5: 4-queens Problem with Constraint Ordering Heuristic.

Figure 6: A continuous optimization problem solved using BT and BT+COH.

Figure 7: System Interface solving and plotting instance <10,40,90>.

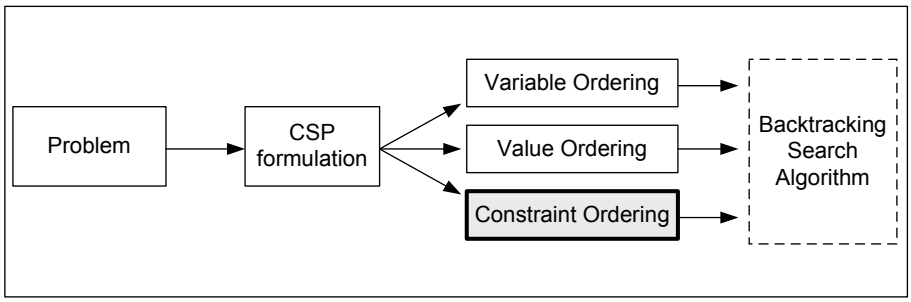


Figure 1. Constraint Ordering

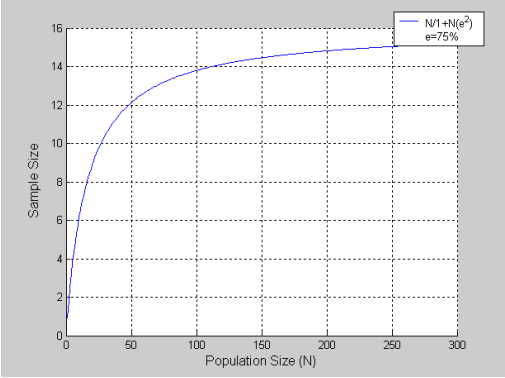


Figure 2. The Yamane's simplified formula for proportions in the 4-queens problem.

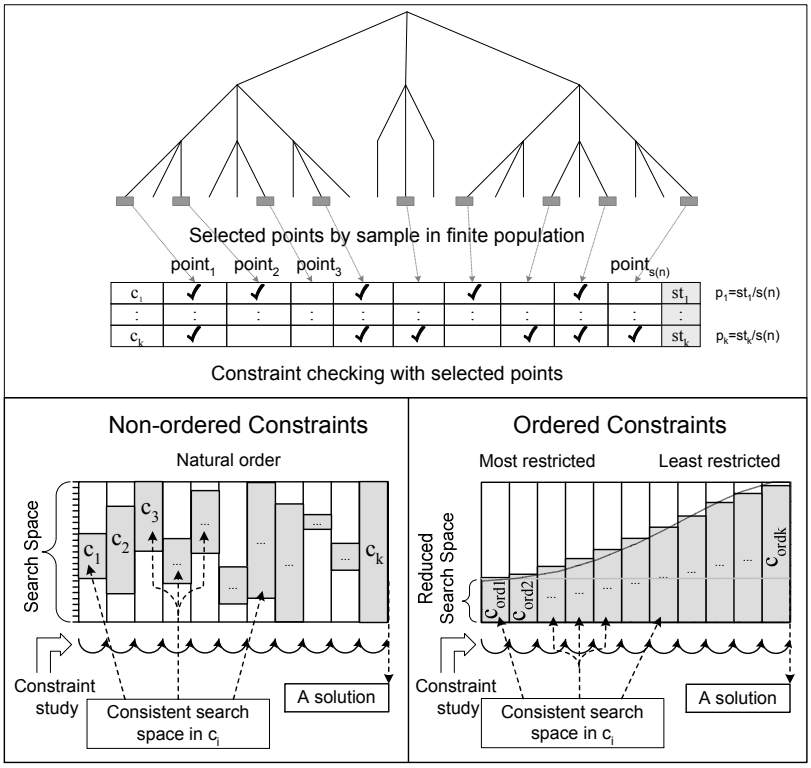


Figure 3. From non-ordered constraint to ordered constraint.

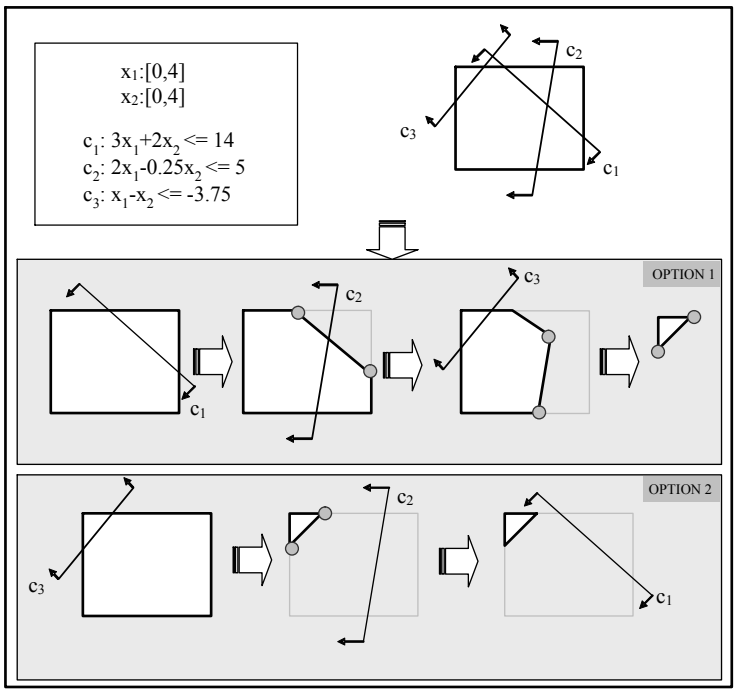
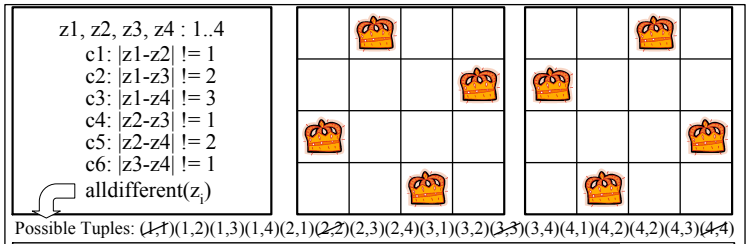


Figure 4. Constraint Ordering Heuristic using HSA.



Possible Tuples: ~~(1,1)~~(1,2)(1,3)(1,4)(2,1)~~(2,2)~~(2,3)(2,4)(3,1)(3,2)~~(3,3)~~(3,4)(4,1)(4,2)(4,2)(4,3)~~(4,4)~~

Natural Order	Constraint Checks: 224	Valid tuples
c_1: $\|z_1-z_2\|!=1$	(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)	6
c_2: $\|z_1-z_3\|!=2$	(1,2)(1,4)(2,3)(3,4)(2,1)(4,1)(3,2)(4,3)	8
c_3: $\|z_1-z_4\|!=3$	(1,2)(1,3)(2,3)(2,4)(3,4)(2,1)(3,1)(3,2)(4,2)(4,3)	10
c_4: $\|z_2-z_3\|!=1$	(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)	6
c_5: $\|z_2-z_4\|!=2$	(1,2)(1,4)(2,3)(3,4)(2,1)(4,1)(3,2)(4,3)	8
c_6: $\|z_3-z_4\|!=1$	(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)	6

Constraint Ordering	Constraint Checks: 168	Valid tuples
c_1: $\|z_1-z_2\|!=1$	(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)	6
c_4: $\|z_2-z_3\|!=1$	(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)	6
c_6: $\|z_3-z_4\|!=1$	(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)	6
c_2: $\|z_1-z_3\|!=2$	(1,2)(1,4)(2,3)(3,4)(2,1)(4,1)(3,2)(4,3)	8
c_5: $\|z_2-z_4\|!=2$	(1,2)(1,4)(2,3)(3,4)(2,1)(4,1)(3,2)(4,3)	8
c_3: $\|z_1-z_4\|!=3$	(1,2)(1,3)(2,3)(2,4)(3,4)(2,1)(3,1)(3,2)(4,2)(4,3)	10

Figure 5. 4-queens Problem with Constraint Ordering Heuristic.

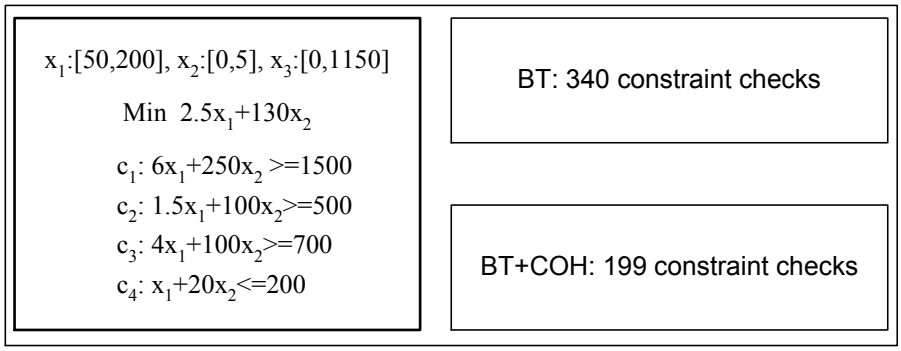


Figure 6. A continuous optimization problem solved using BT and BT+COH.

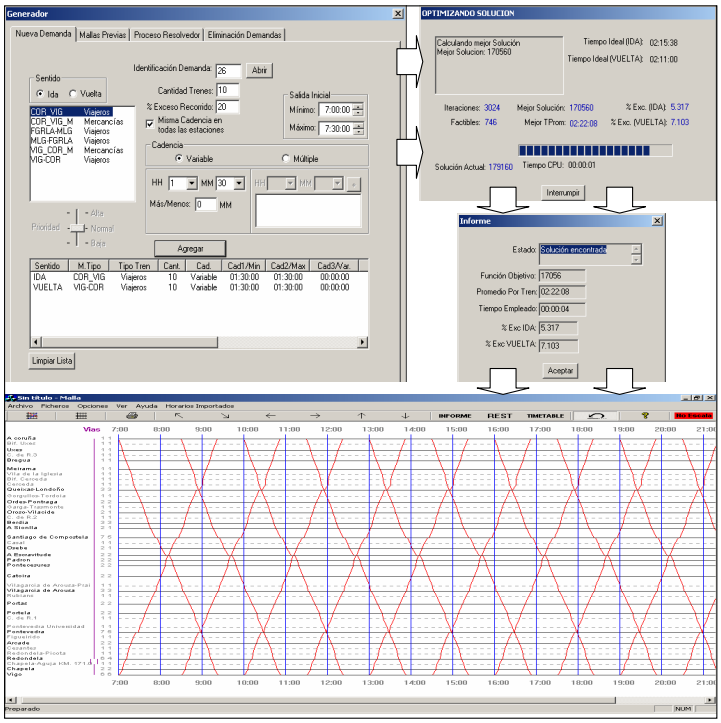


Figure 7. System Interface solving and plotting instance <10,40,90>.

Table 1. Number of constraint checks using Backtracking (BT) and Backtracking with COH (BT+COH)

	Backtracking			**Backtracking+COH**		
instantiations	c_1	c_2	c_3	c_1	c_2	c_3
$(0,0)$	*checked*	*checked*	*checked*	-	-	*checked*
$(0,1)$	*checked*	*checked*	*checked*	-	-	*checked*
$(0,2)$	*checked*	*checked*	*checked*	-	-	*checked*
$(0,3)$	*checked*	*checked*	*checked*	-	-	*checked*
$(0,4)$	*checked*	*checked*	*checked*	*checked*	*checked*	*checked*
$(0,4)$	**Solution**			**Solution**		
$\vdots$	*checked*	*checked*	*checked*			*checked*

Table 2. Number of constraint check saving using our model with GT , BT, FC and RFLA in the n-queens problem.

	GT+COH	**BT+COH**	**FC+COH**	**RFLA+COH**
queens	*Constraint Check Saving*	*Constraint Check Saving*	*Constraint Check Saving*	*Constraint Check Saving*
5	2.1×10^4	2.4×10^2	150	110
10	4.1×10^{11}	3.9×10^7	1.4×10^5	9.3×10^4
20	1.9×10^{26}	3.6×10^{18}	9.6×10^{14}	6.03×10^{11}
50	2.4×10^{70}	3.6×10^{52}	3.1×10^{44}	1.6×10^{32}
100	2.1×10^{143}	2.1×10^{106}	4.5×10^{93}	1.8×10^{66}
150	5.2×10^{219}	3.7×10^{161}	6.8×10^{142}	2.1×10^{100}
200	9.4×10^{295}	8.7×10^{219}	9.9×10^{198}	2.2×10^{134}

Table 3. Runtime and journey time in different problem instances.

	CPLEX				**LINGO**		**TOPOLOGICAL**	
(a) $< n$,40,90>	**Heuristic 1**		**Heuristic 2**		**Heuristic 2**		**Heuristic 2+COH**	
Trains	*runtime*	journey time	*runtime*	journey time	*runtime*	journey time	*runtime*	journey time
5	6”	2:19:48	4”	2:29:33	6”	2:30:54	2”	2:21:54
10	337”	2:20:19	8”	2:26:04	12”	2:31:37	3”	2:22:08
15	601”	2:20:29	12”	2:26:18	19”	2:31:51	3”	2:22:08
20	1065”	2:20:34	16”	2:26:25	25”	2:31:58	3”	2:22:08
50	$> 5h.$	2:20:43	43”	2:31:09	1098”	2:32:11	7”	2:22:08
75	$> 5h.$	2:22:04	$> 1h.$	2:32:14	1590”	2:32:14	11”	2:22:08
(b) <10,s,90>								
10	3”	0:25:06	2”	0:25:06	4”	0:25:06	1”	0:25:06
20	303”	1:04:11	5”	1:04:11	8”	1:04:11	2”	1:04:11
30	$> 1h.$	1:45:38	6”	1:45:08	14”	1:45:38	3”	1:45:08
40	2131”	2:20:10	56”	2:23:36	21”	2:24:36	6”	2:20:22
60	$> 3h.$	3:33:15	217”	3:39:30	180”	3:40:30	15”	3:30:58
(c) <20,40,$f >$								
140	15”	2:16:19	15”	2:20:18	24”	2:16:19	4”	2:18:35
120	156”	2:16:17	14”	2:16:17	23”	2:18:47	4”	2:18:55
100	$> 5h.$	2:22:55	15”	2:23:10	28”	2:22:55	4”	2:19:09
90	1065”	2:20:34	15”	2:26:25	28”	2:31:58	4”	2:22:08
75	$> 1h.$	2:29:18	$> 1h.$	-	25”	2:24:16	6”	2:23:30
60	$> 1h.$	2:21:23	$> 1h.$	-	$> 1h.$	-	46”	2:32:11

Table 4. Random instances $< n, c, d >$, n:variables, c:constraints and d:domain size

Problems	*real constrainedness*	*Parameter* τ	*Number of States*	*Number of Solutions*	*Number of Estimated Sol.*	*% Error*
$< 3, 5, 5 >$	0.09	0.07	125	11.2	8.7	2%
$< 3, 5, 10 >$	0.05	0.043	1000	50	43	0.7%
$< 3, 10, 5 >$	0.024	0.013	125	3	1.6	1.12%
$< 5, 5, 5 >$	0.04	0.038	3125	125	118.7	0.2%
$< 5, 10, 5 >$	0.008	0.01	3125	25	31.2	0.19%
$< 5, 10, 10 >$	0.0045	0.0034	100000	453	340	0.1%

Table 5. Number of constraint checks in problems $< 3, c, 20 >$

		Backtracking	**Backtracking+COH**
problems	*Result*	*constraint checks*	*constraint checks*
$< 3, 3, 20 >$	S	5190.5	1832.1
$< 3, 3, 20 >$	F	27783	9761
$< 3, 5, 20 >$	S	12902.9	2990.8
$< 3, 5, 20 >$	F	46305	9761
$< 3, 7, 20 >$	S	21408.3	3253.2
$< 3, 7, 20 >$	F	64827	9761
$< 3, 9, 20 >$	S	32423.8	4256.5
$< 3, 9, 20 >$	F	83349	9761
$< 3, 11, 20 >$	S	44452.4	4214.3
$< 3, 11, 20 >$	F	101871	9761
$< 3, 13, 20 >$	S	52339.2	4938.5
$< 3, 13, 20 >$	F	120393	9761
$< 3, 15, 20 >$	S	60352.9	5250.7
$< 3, 15, 20 >$	F	138915	9761

Table 6. Number of constraint checks in consistent problems using backtracking filtered with arc-consistency

		Backtracking-AC	**Backtracking-AC+COH**
problems	*Result*	*constraint checks*	*constraint checks*
$< 5, 3, 10 >$	S	2275.5	798.5
$< 5, 5, 10 >$	S	14226.3	2975.2
$< 5, 7, 10 >$	S	35537.4	5236.7
$< 5, 9, 10 >$	S	50315.7	5695.5
$< 5, 11, 10 >$	S	65334	5996.3
$< 5, 13, 10 >$	S	80384	6283.5
$< 5, 15, 10 >$	S	127342	8598.6

Table 7. Number of constraint checks in problems $< 3, 5, d >$

		Backtracking	**Backtracking+COH**
problems	*Result*	*constraint checks*	*constraint checks*
$< 3, 5, 5 >$	S	244.1	55.82
$< 3, 3, 5 >$	F	1080	236
$< 3, 5, 10 >$	S	2544.2	527.8
$< 3, 5, 10 >$	F	6655	1324
$< 3, 5, 15 >$	S	6947.2	1489.4
$< 3, 5, 15 >$	F	20480	4286
$< 3, 5, 20 >$	S	12260.3	2642.1
$< 3, 5, 20 >$	F	46305	10143
$< 3, 5, 25 >$	S	16835.1	3567.1
$< 3, 5, 25 >$	F	87880	17876
$< 3, 5, 30 >$	S	29608.6	5984.1
$< 3, 5, 30 >$	F	148955	29911
$< 3, 5, 35 >$	S	40384.7	8276.9
$< 3, 5, 35 >$	F	233280	46856

Table 8. Number of constraint checks in consistent problems with backtracking filtered with arc-consistency

		Backtracking-AC	**Backtracking-AC+COH**
problems	*Result*	*constraint checks*	*constraint checks*
$< 3, 5, 5 >$	S	93.9	32.7
$< 3, 5, 10 >$	S	180.3	63.06
$< 3, 5, 15 >$	S	241.3	86.26
$< 3, 5, 20 >$	S	320.5	105.1
$< 3, 5, 25 >$	S	419.8	143.9
$< 3, 5, 30 >$	S	514.6	175.9
$< 3, 5, 35 >$	S	655.4	215.1