Distributed Non-binary Constraints

Miguel A. Salido[1] and Federico Barber[2]

[1] Dpto. Ciencias de la Computación e Inteligencia Artificial, Universidad de Alicante
Campus de San Vicente, Ap. de Correos: 99, E-03080, Alicante, Spain
msalido@dsic.upv.es**

[2] Dpto. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia
Camino de Vera s/n, 46071, Valencia, Spain
fbarber@dsic.upv.es

Abstract. Nowadays many real problems can be modeled as Constraint Satisfaction Problems (CSPs). In many situations, it is desirable to be able to state both *hard* constraints and *soft* constraints. Hard constraints must hold while soft constraints may be violated but as many as possible should be satisfied. Although the problem constraints can be divided into two groups, the order in which these constraints are studied can improve efficiency, particulary in problems with non-binary constraints. In this paper, we carry out a classification of hard and soft constraints in order to study the tightest hard constraints first and to obtain ever better solutions. In this way, inconsistencies can be found earlier and the number of constraint checks can be significantly reduced.

1 Introduction

Many problems arising in a variety of domains such as planning, scheduling, diagnosis, decision support, scheduling and design can be efficiently modeled as Constraint Satisfaction Problems (CSPs) and solved using constraint programming techniques. Some of these problems can be modeled naturally using non-binary (or n-ary) constraints. Although, researchers have traditionally focused on binary constraints [9], the need to address issues regarding non-binary constraints has recently started to be widely recognized in the constraint satisfaction literature.

One approach to solving CSPs is to use a depth-first backtrack search algorithm [3]. General methods for solving CSPs include *Generate and test* [7] and *Backtracking* [6] algorithms. Many works have been carried out to improve the *Backtracking* method. One way of increasing the efficiency of *Backtracking* includes the use of *search order* for variables and values. Some heuristics based on *variable ordering* and *value ordering* [5] have been developed, because of the

** This work has been supported by the grant (PPI-02-03) of visitor professor from the Polytechnic University of Valencia, Spain, the grant CTIDIB/2002/112 form the Generalitat Valenciana and the grant DPI2001-2094-C03-03 from the Spanish Government.

additivity of the variables and values. 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 [1].

In spite of the additivity of constraints, only a few works have be done on binary constraint ordering mainly for arc-consistency algorithms [10], [4], but little work has be done on non-binary constraint ordering (for instance in disjunctive constraints [8]), 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 *Hard and Soft Constraint Ordering Heuristic* (HASCOH) that classifies the non-binary constraints, independently of the arity so that hard constraints are studied before soft constraints and then the tightest constraints are studied before the loosest constraints. This is based on the *first-fail* principle, which can be explained as

"To succeed, try first where you are more likely to fail"

HSACOH manages CSPs in a distributed way so that each agent is committed to a set of constraints. The hard constraints that are more likely to fail are studied first using a search algorithm. In this way, inconsistent tuples can be found earlier so that backtrackings are avoided. Without loss of generality, we do not consider preferences in soft constraints, that is, all soft constraints are equally important. Thus, soft constraints are studied after the hard constraints in order to satisfy as many soft constraints as possible. This model allows agents to run concurrently to achieve partial solutions for any-time complete solutions.

2 Preliminaries

CSP: A constraint satisfaction problem (CSP) consists of a set of variables $X = \{x_1, x_2, ..., x_n\}$; a set of finite domains $D = \{D_1, D_2, ..., D_n\}$, where each variable $x_i \in X$ has a set D_i of possible values; and a finite collection of constraints restricting the values that the variables can simultaneously take. We will classify these constraints as hard and soft constraints: hard constraints must hold while soft constraints may be violated, but should be satisfied as much as possible.

State: one possible assignment of all variables; the number of states is equal to the Cartesian product of the domain size.

Partition: A partition of a set C is a set of disjoint subsets of C whose union is C. The subsets are called partition blocks.

Distributed CSP: A distributed CSP is a CSP in which the variables and constraints are distributed among automated agents [11]. Each agent has several variables and attempts to determine their values. However, there are interagent constraints and the value assignment must satisfy these interagent constraints.

Objective in a CSP: A *solution* to a CSP is an assignment of values to all the variables so that at least all the hard constraints are satisfied. Typical tasks of interest are to determine whether a solution exists, to find one or all solutions and to find an optimal or a good solution relative to a preference criterion.

3 Constraint Ordering: An Any-time Proposal

Our main objective is to classify the problem constraints in an appropriate order depending on the desired goals. One way to manage the problem constraints is by means of the natural order in which they are inserted into the problem. However, when managing hard and soft constraints there is a natural and reasonable order where the hard constraints are managed first and the soft constraints are managed later. This natural constraint ordering is presented in Figure 1. Each hard and soft constraint satisfies a portion of the search space, but no ordering is carried out to avoid constraint checking.

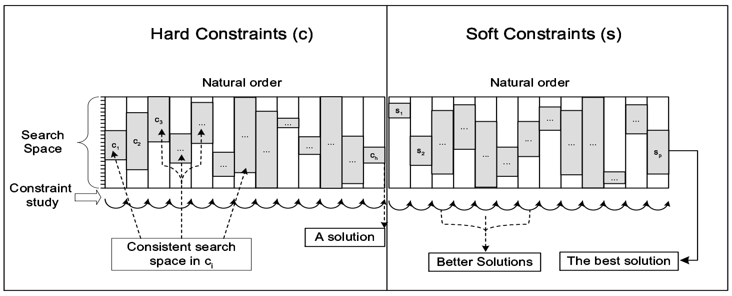


Fig. 1. Natural ordering of hard and soft constraints

In many real problems, the main objective is to obtain a solution that satisfies hard constraints, as soon as possible, and as many soft constraints as possible. In this case, an any-time proposal may be appropriate. A feasible solution may be improved at any time by another solution that satisfies more soft constraints. Thus, both hard and soft constraints are classified from the tightest ones to the loosest ones. This constraint ordering is presented in Figure 2.

The search space of the correctly ordered hard and soft constraints has a behavior which is similar to the behavior of the left tails of normal curves, in which the height of each curve is bounded by the entire search space. The height of the tail of the hard constraints represents the valid search space for the problem. This restricted search space is the only valid search space for finding problem solutions and the rest of the search space can be removed.

Furthermore, the height of the tail of the soft constraints may be zero, because soft constraints are generally over-constrained. However, these constraints are dispensable and the objective is to satisfy as many soft constraints as possible. These soft constraints are classified from the tightest one to the loosest one. Thus, the first solution generated by the study of hard constraints is checked

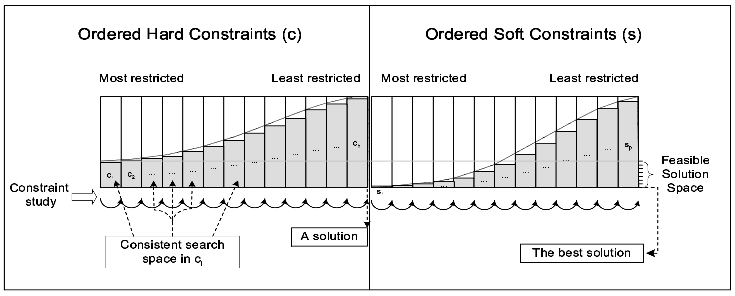


Fig. 2. Constraint ordering in the any-time proposal

with all soft constraints and this solution is labeled with the number of satisfied soft constraints. Due to the any-time behavior, the following solution satisfying all hard constraints is checked with the soft constraints from the tightest one to the loosest one, and this constraint checking is aborted when its label can not be greater than the label of the first solution. Thus, at any time, the best solution is maintained with its label, and a future solution is checked with soft constraints while its label may reach the label of the current best solution.

Here, we will focus on this any-time behavior in which, depending on the user requirements, the solutions can be improved in order to satisfy more soft constraints. Thus our main objective is to classify both hard and soft non-binary constraints in the appropriate order to be solved by some of the current techniques that manage non-binary constraints in a natural way [2].

4 Our Distributed Model: HASCOH

Agent-based computation has been studied for several years in the field of artificial intelligence and has been widely used in other branches of computer science. HASCOH is meant to be a framework for interacting agents to achieve a consistent state. The main idea of our multi-agent model is based on carrying out a partition of the hard constraints, in k groups called *blocks* of constraints, so that the tightest constraints are grouped and studied first by autonomous agents.

To this end, a preprocessing agent carries out a partition of the hard constraints, similar to a sample in finite population, in order to classify both hard and soft constraints from the tightest *hard* constraints to the loosest soft constraints. Then, a group of agents called *hard block agents* concurrently manages each block of hard constraints, generated by the preprocessing agent. Also, an agent called *soft agent* manages all soft constraints. Each *hard block agent* is in charge of solving its partial problem by means of a search algorithm. Thus, a problem solution is incrementally generated from the first *hard block agent* to the last *hard block agent*. Without loss of generality we consider all variables are involved in the hard constraints. Afterwards, the *soft agent* is committed to checking the solutions obtained by the *hard block agents*. Therefore, as an

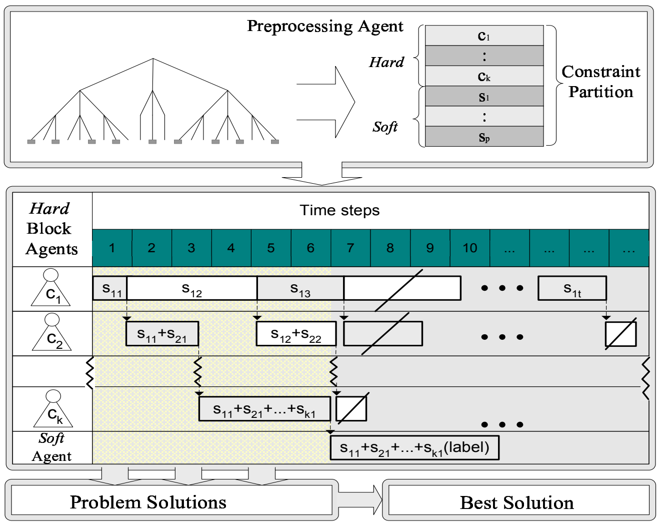


Fig. 3. Multi-agent model

any-time proposal, and depending on the time available, these solutions may be improved by means of the concurrent search in order to find a solution that satisfies as many constraints as possible.

Figure 3 shows the multi-agent model, in which consistent partial states (s_{ij}) are concurrently generated by each *hard block agent* and sent to the following *hard block agent* until a consistent state is found (by the last *hard block agent*). For example, state: $s_{11}+s_{21}+...+s_{k1}$ is a problem solution. Then, the *soft agent* checks this solution, and it is labeled with the number of satisfied soft constraints. We must take into account that a solution is incrementally generated, however partial solutions are concurrently generated due to many partial solutions will not take part in a solution.

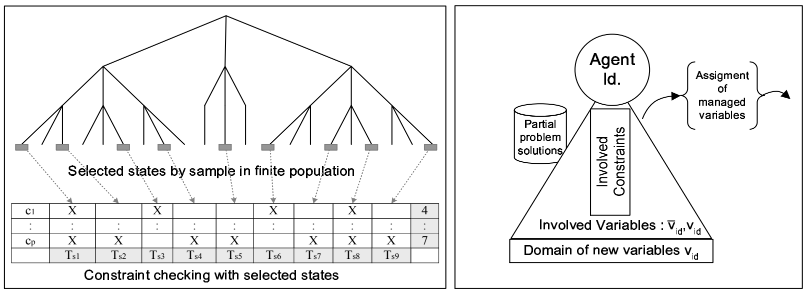
4.1 Preprocessing Agent

The preprocessing agent classifies the constraints in the appropriate order by means of a sample from a finite population in statistics where there is a population, and a sample is chosen to represent this population. In our context, the population is composed by the states generated by means of the Cartesian Product of variable domain bounds and the sample is composed by $s(n)$ random and well distributed states (s is a polynomial function) in order to represent the entire population. As in statistic, the user selects the size of the sample ($s(n)$). The preprocessing agent studies how many states $st_i : st_i \leq s(n)$ satisfy each constraint c_i. Thus, each constraint c_i is labeled with p_i: $c_i(p_i)$, where $p_i = st_i/s(n)$ represents the probability that c_i satisfies the whole problem. Therefore, the computational complexity is $|c|s(n)$. Thus, the preprocessing agent classifies the *hard* constraints in ascending order of the labels p_i and the soft constraints in descending order of the labels p_i. The behavior of a preprocessing agent is shown in Figure 4. It can be observed that a sample of states is selected from the spanning tree. Each state is checked with the constraints and the evaluation value T_{si} may be stored to be used by local search algorithms. Furthermore, each constraint c_i is labeled in order to be classified. Thus, as we pointed in Figure 3, these ordered constraints are partitioned in k blocks (geometrically distributed) to divide the problem in k interdependent subproblems. Each subproblem will be solved by an agent, called *block agent*.

4.2 Hard Block Agent

A *block agent* is a cooperating agent with a set of properties. We make the following assumptions (Figure 4(right)):

- There is a partition of the set of hard constraints $C \equiv \bigcup_{i=1}^{k} C_i$ generated by the preprocessing agent, and each *hard block agent* a_j has a block of constraints C_j.
- Each *hard block agent* a_j knows a set of variables, V_j, which are involved in its block of constraints C_j. These variables fall into different sets: *used variables* set ($\overline{v}_j$) and *new variables* set (v_j), that is: $V_j = \overline{v}_j \cup v_j$.
- The domain D_i corresponding to variable x_i is maintained in the first *hard block agent* a_t in which x_i is involved, (i.e.), $x_i \in v_t$.
- Each *hard block agent* a_j assigns values (by a search algorithm) to variables that have not yet been assigned, that is, a_j assigns values to variables $x_i \in v_j$, because variables $x_k \in \overline{v}_j$ have already been assigned by previous agents $a_1, a_2, ..., a_{j-1}$.
- Each *hard block agent* a_j knows the consistent partial states generated by the previous agents $a_1, a_2, ..., a_{j-1}$. Thus, agent a_j knows assignments of variables included in sets: $\overline{v}_1, \overline{v}_2, ..., \overline{v}_{j-1}$.


Selected states by sample in finite population
Constraint checking with selected states
Agent Id.
Partial problem solutions
Involved Constraints
Assigment of managed variables
Involved Variables : $\overline{v}_{id}, v_{id}$
Domain of new variables v_{id}

c_1	X		X			X		X		4
:	:	:	:	:	:	:	:	:	:	:
c_p	X	X		X	X		X	X	X	7
	T_{s1}	T_{s2}	T_{s3}	T_{s4}	T_{s5}	T_{s6}	T_{s7}	T_{s8}	T_{s9}	

Fig. 4. Behavior of preprocessing agent and hard block agent

Hard block agents cooperate to achieve a consistent state. *Hard block agent* 1 tries to find a consistent state for its partial problem. When it has a consistent partial state, it communicates this partial state to *hard block agent* 2. *Hard block agent* 2 studies the second set of tightest *hard* constraints using the variable assignments generated by *hard block agent* 1. Meanwhile, agent 1 tries to find any other consistent partial state. So, each *hard block agent* j ($j \leq k$), using the variable assignment of the previous hard block agents $1, 2, .., j-1$, tries to concurrently find a more complete assignment. A problem solution is obtained when the last *hard block agent* k finds a consistent state.

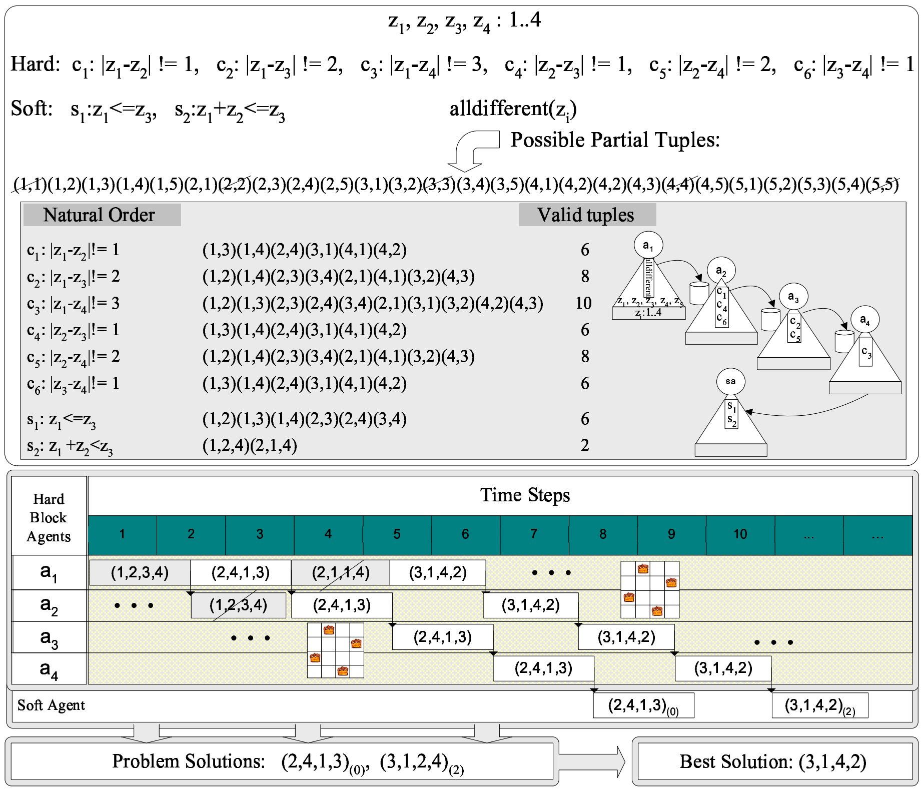
4.3 Soft Agent

Once *hard block agents* find a consistent state, this solution is sent to the *soft agent*. This agent is committed to checking solutions with soft constraints in order to return the best solution at any-time.

The first solution generated by the hard block agents is sent to the *soft agent*. The *soft agent* checks this solution with all soft constraints to evaluate the goodness of this solution. Thus, this solution is labeled with the number of satisfied soft constraints. The second solution generated by the *hard block agents* is also sent to the *soft agent*, and this solution is checked with soft constraints starting from the tightest ones. The constraint checking continues as long as the label of this solution may be greater than the label of the first solution. For instance, if there are ten soft constraints and the first solution satisfies seven soft constraints (its label is 7), the second solution will be checked with the soft constraints (from the tightest to the loosest). When this solution is not consistent with two soft constraints, the soft constraint checking is aborted, because this solution will not satisfy more constraints than the first solution. Thus, any other solution will be checked with soft constraints as long as its label may reach the label of the current best solution.

Example: The 4-queens problem is a classical search problem in the artificial intelligence area. We have extended this problem to include soft constraints. The problem is to place four queens z_1, z_2, z_3, z_4 on a 4×4 chessboard so that no two queens can capture each other. Thus, hard constraints impose the condition that no two queens are allowed to be placed on the same row, the same column, or the same diagonal. We also add two soft constraints: queen 1 value must be less or equal than queen 2 value: $z_1 \leq z_3$ and the sum of queen 1 and queen 2 values must be less or equal than queen 3 value: $z_1 + z_2 \leq z_3$. This modified 4-queens problem is internally managed in Figure 5.

Figure 5 shows the behavior HASCOH. The preprocessing step checks how many partial states (from a given sample: 16 tuples $\{(1,1), (1,2), \cdots, (4,3), (4,4)\}$) satisfy each constraint and classifies them afterwards. It can be observed that some hard constraints are tightest than others. Constraints c_1, c_4, c_6 only satisfy 6 partial states, while constraints c_2 and c_5 satisfy 8 partial states and constraint c_3 satisfies 10 partial states. Furthermore, soft constraint 2 is tightest than soft constraint 1.



z1, z2, z3, z4 : 1..4
Hard: c1: |z1-z2| != 1, c2: |z1-z3| != 2, c3: |z1-z4| != 3, c4: |z2-z3| != 1, c5: |z2-z4| != 2, c6: |z3-z4| != 1
Soft: s1:z1<=z3, s2:z1+z2<=z3
alldifferent(zi)
Possible Partial Tuples:
(1,1)(1,2)(1,3)(1,4)(1,5)(2,1)(2,2)(2,3)(2,4)(2,5)(3,1)(3,2)(3,3)(3,4)(3,5)(4,1)(4,2)(4,2)(4,3)(4,4)(4,5)(5,1)(5,2)(5,3)(5,4)(5,5)
Natural Order
Valid tuples
c1: |z1-z2|!= 1 (1,3)(1,4)(2,4)(3,1)(4,1)(4,2) 6
c2: |z1-z3|!= 2 (1,2)(1,4)(2,3)(3,4)(2,1)(4,1)(3,2)(4,3) 8
c3: |z1-z4|!= 3 (1,2)(1,3)(2,3)(2,4)(3,4)(2,1)(3,1)(3,2)(4,2)(4,3) 10
c4: |z2-z3|!= 1 (1,3)(1,4)(2,4)(3,1)(4,1)(4,2) 6
c5: |z2-z4|!= 2 (1,2)(1,4)(2,3)(3,4)(2,1)(4,1)(3,2)(4,3) 8
c6: |z3-z4|!= 1 (1,3)(1,4)(2,4)(3,1)(4,1)(4,2) 6
s1: z1<=z3 (1,2)(1,3)(1,4)(2,3)(2,4)(3,4) 6
s2: z1 +z2<z3 (1,2,4)(2,1,4) 2
a1
alldifferent
z1, z2, z3, z4, z5
zi:1..4
a2
c1
c4
c6
a3
c2
c5
a4
c3
sa
s1
s2
Hard Block Agents
Time Steps
1
2
3
4
5
6
7
8
9
10
...
...
a1
(1,2,3,4)
(2,4,1,3)
(2,1,1,4)
(3,1,4,2)
• • •
a2
• • •
(1,2,3,4)
(2,4,1,3)
(3,1,4,2)
a3
• • •
(2,4,1,3)
(3,1,4,2)
• • •
a4
(2,4,1,3)
(3,1,4,2)
Soft Agent
(2,4,1,3)(0)
(3,1,4,2)(2)
Problem Solutions: (2,4,1,3)(0), (3,1,2,4)(2)
Best Solution: (3,1,4,2)

Fig. 5. The 4-queens problem in our distributed model.

5 Evaluation of HASCOH

In this section, we compare the performance of our model HASCOH with two well-known and complete CSP solvers: *Generate and Test* (GT) and *Backtracking* (BT), because they are the most appropriate techniques for observing the number of constraint checks. This empirical evaluation was carried out with two different types of problems: benchmark problems and random problems.

Benchmark Problems: The n-queens problem is a classical search problem in the artificial intelligence area. The 4-queens problem was studied in the previous section.

In Table 1, we present the amount of constraint check saving in the n-queens problem using GT with our model (HASCOH+GT) and BT with our model (HASCOH+BT). Here, our objective is to find all solutions. The results show that the amount of constraint check saving was significant in HASCOH+GT and Mod+BT due to the fact that our model classifies the constraints in the appropriate order, so that the tightest constraints were checked first, and inconsistent tuples were discarded earlier.

Table 1. Number of constraint check saving using our model with GT and BT in the n-queens problem.

	HASCOH+GT	**HASCOH+BT**
queens	*Constraint Check Saving*	*Constraint Check Saving*
5	2.1×10^{4}	2.4×10^{2}
10	4.1×10^{11}	3.9×10^{7}
20	1.9×10^{26}	3.6×10^{18}
50	2.4×10^{70}	3.6×10^{52}
100	2.1×10^{143}	2.1×10^{106}
150	5.2×10^{219}	3.7×10^{161}
200	9.4×10^{295}	8.7×10^{219}

Random Problems: Benchmark sets are used to test algorithms for specific problems. However, 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,number of constraints, domain size, arity of constraints, etc). 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 our empirical evaluation, each set of random constraint satisfaction problems was defined by the 4-tuple $< n, c, s, d >$, where n was the number of variables, c the number of hard constraints, s the number of soft 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. Thus, Table 2 sets three 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 2. Number of constraint checks using Backtracking filtered with Arc-Consistency.

	BT-AC	**HASCOH+BT-AC**		**BT-AC**	**HASCOH+BT-AC**
problems	*constraint checks*	*constraint checks*	*problems*	*constraint checks*	*constraint checks*
$< 5, 5, 5, 10 >$	14226.5	2975.5	$< 3, 5, 5, 10 >$	150.3	33.06
$< 5, 10, 5, 10 >$	60250.3	5714.2	$< 3, 5, 5, 20 >$	260.4	55.2
$< 5, 20, 5, 10 >$	203542.2	12548.5	$< 3, 5, 5, 30 >$	424.3	85.26
$< 5, 30, 5, 10 >$	325487.4	17845.7	$< 3, 5, 5, 50 >$	970.5	180.1
$< 5, 50, 5, 10 >$	513256.7	24875.5	$< 3, 5, 5, 70 >$	2104.8	380.9
$< 5, 75, 5, 10 >$	704335.1	34135.3	$< 3, 5, 5, 90 >$	4007.4	701.7
$< 5, 100, 5, 10 >$	895415.3	43396.6	$< 3, 5, 5, 110 >$	7851.4	1205.1

The number of constraint checks using BT filtered by *arc-consistency* (as a preprocessing) (BT-AC) and BT-AC using our model (HASCOH+BT-AC) is presented in Table 2. On the left side of the table, we present the number of constraint checks in problems where the number of hard constraints was increased from 5 to 100 and the number of variables, soft constraints and the domain size were set at 5,5 and 10, respectively: $< 5, c, 5, 10 >$. The results show that the number of constraint checks was reduced in all cases. On the right side of the table, we present the number of constraint checks in problems where the domain size was increased from 10 to 110 and the number of variables, the number of hard constraints and the number of soft constraints were set at 3,5 and 5, respectively: $< 3, 5, 5, d >$. The results were similar and the number of constraint checks was also reduced in all cases.

6 Conclusions and Future Work

In this paper, we propose a distributed model for solving Constraint Satisfaction Problems (CSPs) in which agents are committed to solving their partial problems by means of search algorithms. The solutions are incrementally created by each hard block agent in order to satisfied the hard constraints and as many soft constraints as possible. Hard and soft constraints are ordered to reduce the number of constraint checks.

As future work, we are working on a distributed model in which *block agents* can dynamically interchange constraints, depending on the evaluation values, so that the preprocessing agent can be removed and block agents can carry out this constraint partition.

References

1. R. Barták, 'Constraint programming: In pursuit of the holy grail', *in Proceedings of WDS99 (invited lecture), Prague, June*, (1999).
2. C. Bessière, P. Meseguer, E.C. Freuder, and J. Larrosa, 'On forward checking for non-binary constraint satisfaction', *Artifical Intelligence*, 205–224, (2002).
3. J.R. Bitner and Reingold E.M., 'Backtracking programming techniques', *Communications of the ACM 18*, 651–655, (1975).
4. I.P. Gent, E. MacIntyre, P. Prosser, and T Walsh, 'The constrainedness of arc consistency', *Principles and Practice of Constraint Programming*, 327–340, (1997).
5. R.M. Haralick and Elliot G.L., 'Increasing tree search efficiency for constraint satisfaction problems', *Artificial Intelligence*, **14**, 263–313, (1980).
6. V. Kumar, 'Depthfirst search', *In Encyclopedia of Artificial Intelligence*, **2**, 1004–1005, (1987).
7. V. Kumar, 'Algorithms for constraint satisfaction problems: a survey', *Artificial Intelligence Magazine*, **1**, 32–44, (1992).
8. M.A. Salido and F. Barber, 'A polynomial algorithm for continuous non-binary disjunctive CSPs: extended DLRs', *Knowledge-Based Systems, In press*, **16**, (2003).
9. E. Tsang, *Foundation of Constraint Satisfaction*, Academic Press, 1993.
10. R. Wallace and E. Freuder, 'Ordering heuristics for arc consistency algorithms', *In Proc. of Ninth Canad. Conf. on A.I.*, 163–169, (1992).
11. M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara, 'The distributed constraint satisfaction problem: Formalization and algorithms', *Knowledge and Data Engineering*, 673–685, (1998).