Optimization-based Heuristics for Maximal Constraint Satisfaction*

Javier Larrosa and Pedro Meseguer†

Universitat Politècnica de Catalunya
Dep. Llenguatges i Sistemes Informàtics
Pau Gargallo 5, 08028 Barcelona, SPAIN
E-mail: {larrosa,meseguer}@lsi.upc.es

Abstract. We present a new heuristic approach for maximal constraint satisfaction of overconstrained problems (MAX-CSP). This approach is based on a formulation of CSP as an optimization problem presented in a previous paper [Meseguer and Larrosa, 95], which has given good results on some classes of solvable CSP. For MAX-CSP, we have developed two heuristics for dynamic variable and value ordering, called highest weight and lowest support respectively, to be used inside the extended forward checking algorithm (P-EFC3). These heuristics are expensive to compute, so we have developed an incremental updating formula to avoid redundant computation. We have tested both heuristics with the P-EFC3 algorithm on several instances of two classes of random CSP. Experimental results show that both heuristics outperform previously used heuristics based on inconsistency counts. In fact, the lowest support heuristic appears as a kind of generalization of these previous heuristics, including extra information about future variables.

1 Introduction

Constraint satisfaction problems (CSP) consider the assignment of values to variables under a set of constraints. When a total assignment satisfying every constraint exists, this assignment is a solution and the problem is said to be solvable. If such an assignment does not exist, the problem is said to be overconstrained or unsolvable, and the best one can obtain is a total assignment satisfying as many constraints as possible. This assignment is called a partial solution satisfying a maximal number of

* This research has been supported by the Spanish CICYT under the project #TAP93-0451.

† Current address: Institut d'Investigació en Intel.ligència Artificial, CSIC, Campus UAB, 08193 Bellaterra, SPAIN.

constraints. This specific problem is called the maximal constraint satisfaction problem (usually abbreviated as MAX-CSP), and it is of interest in several areas of application [Fox, 87], [Feldman and Golumbic, 90], [Bakker *et al.*, 93].

The purpose of this paper is to show the applicability of a new heuristic approach to MAX-CSP. This approach is based on an optimization formulation to solve CSP contained in a previous paper [Meseguer and Larrosa, 95], where we formulated the problem of discrete constraint satisfaction as an optimization problem. Specifically, we provided a way to construct, for any binary and discrete CSP, a continuous function A(P) whose global maximum corresponds to the best solution for the CSP. By the best solution we mean either (i) a global solution satisfying every constraint, if the problem is solvable, or (ii) a partial solution satisfying a maximal number of constraints, if the problem is overconstrained. Therefore, the best solution for any CSP can be obtained constructing the function $A(P)$ and computing its global maximum using any kind of standard optimization techniques. Global optimization techniques present several drawbacks: their applicability depends on the kind of function to optimize and they require significant computational resources. On the other hand, local optimization methods can easily compute a maximum but there is no guarantee that it will be a global one. To circumvent all these difficulties, we developed heuristics for variable and value selection which use information from the $A(P)$ function. Using these heuristics in a forward checking algorithm, we obtained good results for several classes of solvable CSP.

Following the same approach for MAX-CSP, in this paper we present two heuristics called *highest weight* and *lowest support*, to be used inside the extended forward checking algorithm (P-EFC3) [Freuder and Wallace, 92]. These heuristics provide dynamic variable and value ordering. They are quite well informed, but expensive to compute. To overcome this drawback, we have developed an incremental updating formula which avoids the repetition of redundant operations. These heuristics with P-EFC3 outperform previously used heuristics based on inconsistency counts on several classes of overconstrained CSP.

The paper is organized as follows. In section 2 we provide a brief summary of previous work on MAX-CSP. We outline our formulation of CSP as optimization problems in section 3. Based on this formulation, we describe the highest weight and lowest support heuristics in section 4. The incremental heuristic computation appears in section 5. Experimental results of the execution of P-EFC3 with and without these heuristics appear in section 6, and we discuss these results in section 7. Finally, section 8 contains the conclusions of this work.

2 Related Work

The simplest algorithm for maximal constraint satisfaction follows a *branch and bound* scheme. This algorithm performs a systematic traversal on the search tree

generated by the problem, associating with every —partial or total— assignment of variables a cost function, which is the number of violated constraints caused by this assignment. This cost function is usually called the *distance* from a global solution, that is, the solution satisfying every constraint which does not exist for overconstrained CSP. Branch and bound keeps track of the best solution obtained so far, which is the total assignment with minimal distance in the explored part of the search tree. When a partial solution has a distance greater than or equal to the distance of the current best solution, this line of search is abandoned because it cannot lead to a better solution than the current one. In this way, the distance of the current best solution is used as an *upper bound* of the allowable cost, while the distance of the current partial solution is a *lower bound* of the cost for any solution including this partial solution.

The basic branch and bound can be enhanced with more sophisticated strategies based on previous work on solvable CSP. *Prospective* algorithms look ahead to compute some form of local consistency among future variables. The most common prospective algorithm is *forward checking*, which evaluates the impact of the current partial solution on future variables, recording for each possible value the number of violated constraints with the current partial solution (inconsistency-counts). This improves the lower bound of the current partial solution. *Retrospective* algorithms remember previous actions in order to avoid repeating them in the future and, in this way, they can save redundant constraint checks. The most familiar retrospective algorithms are *backjumping* and *backmarking*. All these algorithms guarantee that the optimal solution will be found. For a detailed description, see [Freuder and Wallace, 92].

Several heuristics for variable and value selection have been developed. For static variable ordering, the most promising heuristic orders variables by decreasing width. This heuristic is enhanced when combined with a second heuristic to break ties, such as minimum domain size, maximum degree, or largest mean ACC (arc consistency counts) in its domain. These combinations are called conjunctive width heuristics [Wallace and Freuder, 93]. For static value ordering, values are ordered by increasing ACC. Regarding dynamic variable ordering in forward checking, variables are ordered either by the largest mean of inconsistency counts in their domains or by minimum domain size. Dynamic value ordering considers values by increasing inconsistency counts [Freuder and Wallace, 92].

Finally, a strategy to improve the computation of the lower bound for the current partial solution with fixed variable order is proposed in [Wallace, 94]. In a preprocessing step, DAC (directed arc consistency counts) are computed for each value following the fixed order. DAC counts are added to other counts (distance and inconsistency counts) to compute a better lower bound for the current partial solution.

3 An Optimization Approach to CSP

A discrete binary CSP is defined by a finite set of variables $\{X_i\}$ taking values on discrete and finite domains $\{D_i\}$ under a set of binary constraints $\{R_{ij}\}$. A constraint R_{ij} is a logical expression which evaluates to true or false on each pair of potential values for X_i and X_j. The number of variables is n and, without loss of generality, we will assume a common domain D for all the variables, m being its cardinality. A global solution of the CSP is an assignment of values to variables satisfying every constraint. If no solution exists the CSP is overconstrained; in this case we are interested in finding solutions satisfying a maximal number of constraints. This problem is usually referred as MAX-CSP.

A solvable or overconstrained CSP can be formulated as an optimization problem as follows. Given a CSP, we define a *weighted labeling* P as a vector of the space K defined by,

$$K = \{\, P \in R^{nm} \mid P = [p_1,\ldots, p_n];\ p_i = [p_i[\lambda_1],\ldots,p_i[\lambda_m]] \in R^m;$$
$$0 \le p_i[\lambda_k] \le 1\ ;\ \sum_{k=1}^{m} p_i[\lambda_k] = 1,\ i=1,\ldots,n\ \}$$

P contains n components p_i, each corresponding to a variable X_i. A component p_i is a vector of m components, each corresponding to a different value $\lambda_k \in D$. A single component $p_i[\lambda_k]$ is the weight associated with the value λ_k of variable X_i. A weight can take a value between 0 and 1, and the sum of the weights associated with the values of a variable must be 1. An *unambiguous labeling* is a weighted labeling for which the condition $0 \le p_i[\lambda_k] \le 1$ is substituted by $p_i[\lambda_k] \in \{0,1\}$. The space of unambiguous labelings is denoted by K^*. There is a bijection between the set of the total assignments of values to variables and the set of unambiguous labelings: for every variable X_i assigned to a value λ_k the corresponding labeling takes $p_i[\lambda_k]=1$ and $p_i[\lambda]=0$ for all $\lambda \in D$, $\lambda \neq \lambda_k$, and vice versa. For this reason, we will use unambiguous labelings and total assignments as interchangeable. An *ambiguous labeling* is a weighted labeling that is not unambiguous. We define the set $\{r_{ij}\}$ of compatibilities as follows,

$r_{ij}(\lambda_k,\lambda_l) = 1$ if $R_{ij}(\lambda_k,\lambda_l) =$ true
$r_{ij}(\lambda_k,\lambda_l) = -1$ if $R_{ij}(\lambda_k,\lambda_l) =$ false

Compatibility $r_{ij}(\lambda_k,\lambda_l)$ exists for every pair of variables (X_i, X_j). If X_i does not constrain X_j, $r_{ij}(\lambda_k,\lambda_l) = 1$ for all $\lambda_k, \lambda_l \in D$.

Given a labeling P, the *support* that a value λ_k at variable X_i obtains from variable X_j is,

$$s_i(\lambda_k,X_j) = \sum_{l=1}^{m} r_{ij}(\lambda_k,\lambda_l) p_j[\lambda_l] \qquad (1)$$

and the support that a value λ_k at variable X_i obtains from the whole labeling P is,

$$s_i(\lambda_k,P) = \sum_{j=1}^{n} \sum_{l=1}^{m} r_{ij}(\lambda_k,\lambda_l) p_j[\lambda_l]$$

We define the *average local consistency function* $A(P)$ as follows,

$$A(P) = \sum_{i=1}^{n} \sum_{k=1}^{m} \sum_{j=1}^{n} \sum_{l=1}^{m} r_{ij}(\lambda_k,\lambda_l)\ p_i[\lambda_k]\ p_j[\lambda_l] \qquad (2)$$

The following result allows us to formulate a CSP as an optimization problem:

Theorem [Meseguer and Larrosa, 95]: Let us consider a binary CSP and let P_0 be an unambiguous labeling. P_0 violates a minimal number of constraints if and only if P_0 is a global maximum of $A(P)$.

This theorem gives a necessary and sufficient condition for the best possible solution one can compute for a CSP: it must be a global maximum of $A(P)$. By best possible solution we mean either (i) a global solution satisfying every constraint, if the problem is solvable, or (ii) an assignment satisfying a maximal number of constraints, if the problem is overconstrained. The previous theorem is restricted to unambiguous labelings; however, maximizing $A(P)$ we can find a global maximum P_0 which is an ambiguous labeling. In this case, there exists a theorem [Sastry and Thathachar, 94] which assures the existence of an unambiguous labeling P_1 such that $A(P_0) = A(P_1)$, that is, P_1 is also a global maximum and it is unambiguous.

The previous theorem allows us to formulate any binary CSP (no matter whether it is solvable or overconstrained) as an optimization problem, and to solve it by standard optimization methods. However, the theorem needs to compute the global maximum of $A(P)$, which is not an easy task for continuous non-convex functions (the case of $A(P)$ in general). There is no standard approach to global optimization: different global optimization methods exist [Horst and Tuy, 93], their applicability depends on specific features of the function to be optimized and they require significant computational resources. Without discarding the possibility of computing the global maximum of $A(P)$ by global optimization methods, which requires further investigation, we considered the applicability of local optimization techniques in the particular case of solvable CSP [Meseguer and Larrosa, 95]. We used the projected gradient algorithm which computes a new point P^{new} from a point P in the following way,

$$P^{new} = P + \alpha \text{ Proj } (Q) \qquad (3)$$

where Q is the gradient of $A(P)$ on P, the operator Proj is the projection on the set K and α is the step size such that $A(P) < A(P^{new})$. We had to use the projected gradient algorithm instead of the pure gradient because this is a case of constrained

optimization, where the new generated point should lie on the set K. This approach was feasible for solvable CSP, because solutions can be easily identified when they are found (the number of violated constraints is equal to 0). However, this approach is no longer valid for MAX-CSP because we do not know the value of $A(P)$ on its global maximum, or equivalently, we do not know the number of violated constraints in a maximal consistent solution. To overcome these difficulties, we developed a heuristic approach which is explained in the next section.

4 Dynamic Variable and Value Ordering Heuristics

To circumvent the difficulties of local optimization methods when applied to MAX-CSP, we decided to use a systematic search algorithm —extended forward checking— to guarantee that we will find solutions which will be maximally consistent. To guide the search process, we use information provided by the optimization approach. We have developed two heuristics based on this approach. These heuristics provide advice for dynamic variable and value ordering. They are called the *highest weight* heuristic and the *lowest support* heuristic.

To compute these heuristics, we have to relate systematic search with label updating. At a given time, the current state of past and future variables is reflected in a labeling P in the following form:

if X_i is a past variable	$p_i[\lambda^i] = 1$,	λ^i is assigned to X_i
	$p_i[\lambda] = 0$,	$\lambda \in D$, $\lambda \neq \lambda^i$
if X_i is a future variable	$p_i[\lambda] = 1/m_i$,	$\lambda \in feasible(D, X_i)$, $m_i = card(feasible(D, X_i))$
	$p_i[\lambda] = 0$,	$\lambda \in D - feasible(D, X_i)$

where the set $feasible(D, X_i)$ is the set of values that, at that point of the search, are still feasible for the variable X_i. This labeling is unambiguous in past variables and ambiguous in future ones, with a homogeneous weight distribution among the feasible values of future variables.

4.1 The Highest Weight Heuristic

The highest weight heuristic follows a hill-climbing approach. An iteration of the projected gradient algorithm (3) maximizing $A(P)$ causes changes in weights associated with variable-value pairs, producing a new labeling P^{new} such that $A(P) < A(P^{new})$. Those weights of P^{new} which have increased with respect to P correspond to directions in which $A(P)$ locally increases. At this point, we use these weights for variable and value selection. The highest weight heuristic works as follows:

1. Before a variable is assigned, a single iteration of the projected gradient algorithm is performed, starting from the labeling P corresponding to

the current search state.

2. P^{new} is used to provide heuristic advice for variable and value selection:
   - Variable selection: selects the variable with the highest weight associated with a value of its domain (breaking ties randomly).
   - Value selection: selects the value corresponding to the highest weight in the domain of the variable (breaking ties randomly).

This heuristic selects the variable X_i with the "most promising value". The selected variable-value pair defines a direction in which $A(P)$ increases. Once a variable X_i has been selected, all their values will be considered by decreasing weight. This means that the subspace of K^* associated with the variable X_i will be completely explored, considering first those directions in which $A(P)$ increases more.

To compute the new labeling P^{new}, we replace the projected gradient algorithm (3) by the following formula for relaxation labeling [Rosenfeld *et al.*, 76],

$$p_i^{new}[\lambda_k] = \frac{p_i[\lambda_k](2n+q_i[\lambda_k])}{\sum_{k=1}^{m} p_i[\lambda_k](2n+q_i[\lambda_k])} \qquad (4)$$

which has been proved to be an approximation of the projected gradient [Hummel and Zucker, 83], and it is less costly to compute. The elements $q_i[\lambda_k]$ are the components of Q, the gradient vector of $A(P)$ on the point P, which has the following expression [Hummel and Zucker, 83],

$$q_i[\lambda_k] = 2 \sum_{j=1}^{n} \sum_{l=1}^{m} r_{ij}(\lambda_k,\lambda_l) p_j[\lambda_l] \; = 2\, s_i(\lambda_k,P) \qquad (5)$$

4.2 The Lowest Support Heuristic

The lowest support heuristic is inspired by the fail-first principle. A component $q_i[\lambda_k]$ of Q, the gradient of $A(P)$, is proportional to the support $s_i(\lambda_k,P)$ (5). The support $s_i(\lambda_k,P)$ is an averaged sum of compatibilities between the value λ_k of variable X_i and the labeling P; the lower this support is, the more incompatible appears λ_k for X_i with the labeling P. The sum of gradient components for variable X_i, $\sum q_i[\lambda]$, $\lambda \in feasible(D, X_i)$, is a global measure of the support that the set of feasible values of X_i obtains from all the other variables. The variable with the lowest sum of gradient components appears as the variable with the most incompatible set of values with respect to the labeling P. Following the fail-first principle, this variable is a good candidate to be selected. The lowest support heuristic is based on this fact. This heuristic works as follows,

1. Before a variable is assigned, the gradient Q is computed from the labeling P corresponding to the current search state.

2. Q is used to provide advice for variable and value selection:
   - Variable selection: selects the variable with the lowest sum of gradient components (breaking ties randomly).
   - Value selection: selects the value corresponding to the highest gradient component (breaking ties randomly).

This heuristic selects a variable with a small denominator in (4), which —depending on the numerator value— can produce a high value for a weight associated with a value of its domain, although it is not necessarily the highest. Once the variable has been selected, values are considered by decreasing gradient component, or equivalently, by decreasing weight.

5 Incremental Heuristic Computation

We have tested the proposed heuristics inside the extended forward checking algorithm P-EFC3 [Freuder and Wallace, 92], because it permits dynamic variable and value ordering. A version of the P-EFC3 algorithm is given in Fig. 1. In addition, P-EFC3 enhanced with dynamic variable ordering (the largest mean of inconsistency counts) and value ordering (increasing inconsistency counts), is considered one of the most efficient algorithms for MAX-CSP, specially for dense constrained problems.

```
procedure P-EFC3 (current_solution, distance, remaining_variables, remaining_domains)
/*  global variables: best_solution  and  best_distance  */

1   v := select-next-variable(remaining_variables);  /* depends on variable ordering */
2   values := sort-values(v, remaining_domains);   /* depends on value ordering */
3   while values ≠ Ø do
4         l := first(values);
5         new_distance := distance + incons_count(v, l);
6         if remaining_variables = Ø then
7                   if new_distance < best_distance then
8                             best_distance := new_distance;
9                             best_solution := current_solution + (v, l);
10                  endif
11        else
12                  if (new_distance + sum_min_incons_counts < best_distance) then
13                            new_remaining_domains:= update_incons_counts(domains,v,l);
14                            if (new_distance + sum_min_incons_counts < best_distance) then
15                                  P-EFC3 (current_solution + (v, l), new_distance,
16                                            remaining_variables - v, new_remaining_domains);
17                            endif
18                  endif
19        endif
20  values := values - l;
```

```
21 endwhile
endprocedure
```

Fig. 1. The extended forward checking algorithm.

Regarding the proposed heuristics, both are expensive to compute. To make them useful, their computational cost must not surpass the benefits that they produce in searching. Otherwise, their use would be counterproductive. With this goal, we have developed simpler updating formulas, described below, which supersede (4) and (5) and allow us a much more efficient computation of weights and gradients. However, to employ these new formulas we cannot remove values in variable domains. This is the only difference between our implementation and the pure P-EFC3 algorithm, and its consequences are discussed in section 7. In the following, we provide a detailed explanation about how weights and gradients are computed, assuming that the highest weight heuristic —which requires a more complex calculation— is used for variable and value selection.

5.1 Computing Weights

As we explained in section 4, we have to relate systematic search with label updating. For each state s of the search, we define a labeling $P^{(s)}$ as follows,

if X_i is a past variable,	$p_i^{(s)}[\lambda^i] = 1$,	λ^i is assigned to X_i in state s
	$p_i^{(s)}[\lambda] = 0$,	$\lambda \in D$, $\lambda \neq \lambda^i$
if X_i is a future variable	$p_i^{(s)}[\lambda] = 1/m$,	$\lambda \in D$

where we assume that no value is discarded in future variable domains in any step of the algorithm. We use (4) to compute the new labeling $P^{(s\text{-}new)}$, which will be used to select the next variable. If the new variable is assigned, we move to another state s' defined by the state s plus this variable assignment. Otherwise, if no value can be assigned to the new variable, a backtracking occurs and we return to a previous state.

If X_i is a past variable it is easy to see that (4) does not change its weights, that is, $p_i^{(s\text{-}new)}[\lambda] = p_i^{(s)}[\lambda]$ for every $\lambda \in D$. Therefore, we will execute (4) on future variables only, obtaining the following updating formula,

$$p_i^{(s\text{-}new)}[\lambda_k] = \frac{p_i^{(s)}[\lambda_k](2n+q_i^{(s)}[\lambda_k])}{\sum_{k=1}^{m} p_i^{(s)}[\lambda_k](2n+q_i^{(s)}[\lambda_k])} =$$

$$= \frac{1/m\ (2n+q_i^{(s)}[\lambda_k])}{\sum_{k=1}^{m} 1/m\ (2n+q_i^{(s)}[\lambda_k])} = \frac{2n+q_i^{(s)}[\lambda_k]}{2nm+\sum_{k=1}^{m} q_i^{(s)}[\lambda_k]} \qquad (6)$$

where $q_i^{(s)}[\lambda_k]$ are the components of the vector $Q^{(s)}$, the gradient of $A(P)$ on the point $P^{(s)}$.

5.2 Computing Gradients

From (5), we can express $Q^{(s)}$ as a combination of contributions of past and future variables. Let $\boldsymbol{P}$ and $\boldsymbol{F}$ be the sets of indexes of past and future variables at the state s, the gradient $Q^{(s)}$ can be expressed as,

$$q_i^{(s)}[\lambda_k] = q_i^{P(s)}[\lambda_k] + q_i^{F(s)}[\lambda_k] \tag{7}$$

$$q_i^{P(s)}[\lambda_k] = 2 \sum_{j \in \boldsymbol{P}} \sum_{l=1}^{m} r_{ij}(\lambda_k,\lambda_l) p_j^{(s)}[\lambda_l]$$

$$q_i^{F(s)}[\lambda_k] = 2 \sum_{j \in \boldsymbol{F}} \sum_{l=1}^{m} r_{ij}(\lambda_k,\lambda_l) p_j^{(s)}[\lambda_l]$$

Components $q_i^{P(s)}[\lambda_k]$ and $q_i^{F(s)}[\lambda_k]$ are called the past and the future components of the gradient, respectively. It is easy to see that,

$$q_i^{P(s)}[\lambda_k] = 2 \sum_{j \in \boldsymbol{P}} r_{ij}(\lambda_k,\lambda^j) \tag{8}$$

since for a past variable X_j every weight is 0 but the weight associated with the assigned value λ^j which is 1.

After computing $P^{(s\text{-}new)}$ from $Q^{(s)}$ by (6), the algorithm takes a decision in terms of assigning a variable, and it produces a new state s' which has associated the labeling $P^{(s')}$. After this decision, we update $Q^{(s)}$ to obtain the gradient corresponding to s', that is the vector $Q^{(s')}$. In this way, we perform an incremental updating of the gradient vector, computing $Q^{(s')}$ from the previous gradient $Q^{(s)}$ plus the assignment which has occurred between s and s'. We save computational resources, since only those operations strictly needed are performed, in contrast with the computation from scratch of $Q^{(s')}$ using (5).

Gradient updating is as follows. Providing the change from s to s' is the assignment of the value λ^r to the variable X_r, this variable is no longer a future variable and it becomes a past variable. $Q^{(s')}$ can be computed from $Q^{(s)}$ by subtracting from the future component of the gradient the contribution of X_r when it was a future variable, and adding to the past component of the gradient the current contribution of X_r as a past variable,

$$q_i^{F(s')}[\lambda_k] = q_i^{F(s)}[\lambda_k] - 2 \sum_{l=1}^{m} r_{ir}(\lambda_k,\lambda_l)\, p_r^{(s)}[\lambda_l] = q_i^{F(s)}[\lambda_k] - 2\, s_i^{(s)}(\lambda_k,X_r) \tag{9}$$

$$q_i^{P(s')}[\lambda_k] = q_i^{P(s)}[\lambda_k] + 2\, r_{ir}(\lambda_k,\lambda^r) \tag{10}$$

where $s_i^{(s)}(\lambda_k,X_r)$ is the support that the values of the variable X_r bring to the value λ_k for the variable X_i in labeling $P^{(s)}$ (see (1)). Therefore, using (9) and (10) we can easily compute $Q^{(s')}$ from $Q^{(s)}$ and the assignment from $P^{(s)}$ to $P^{(s')}$. In terms of the algorithm P-EFC3 of Fig. 1, weight computation is performed in line 1 using (6), inside the function select_next_variable. To compute the gradient, formula (7) is used adding the past and future components which are always kept updated. Thus, after a variable assignment, the gradient components are updated in line 15 before the recursive call, using (9) and (10).

6 Experimental Results

The proposed heuristics have been tested on two classes of randomly generated CSP, to which we will refer as *fixed tightness* and *variable tightness*. A fixed tightness random CSP is characterised by a 4-tuple $<n, m, p_1, p_2>$ [Prosser, 94], where n is the number of variables, m is the common cardinality of their domains, p_1 is the probability of a constraint existing between a pair of variables (connectivity) and p_2 is the probability of a conflict occurring between two values assigned to constrained variables (tightness). Fixed tightness implies constraint homogeneity: every pair of constrained variables has the same expected number of conflicts, p_2*m^2, independently of the variables considered. This homogeneity can be seen as unnatural for the representation of some real CSP.

A variable tightness random CSP is characterised by a 5-tuple $<n, m, p_1, p_2^{inf}, p_2^{sup}>$, where n, m and p_1 are defined as above, and p_2^{inf} and p_2^{sup} are the tightness lower and upper bound, respectively. For each constraint, its tightness is randomly chosen within the interval $[p_2^{inf}, p_2^{sup}]$. Variable tightness allows some diversity in the constraints: every pair of constrained variables does not have the same expected number of conflicts, which varies from $p_2^{inf}*m^2$ to $p_2^{sup}*m^2$.

6.1 Fixed Tightness Random CSP

Our experiments were performed with $<10, 10, p_1, p_2>$ CSP, with p_1 taking values 0.6, 0.8 and 1.0, and p_2 ranging from 0.5 to 0.9. Twenty instances of each parameter setting were created forming a set of 300 problems divided into 15 sets. Because of the values of p_1 and p_2, these problems are typically unsolvable. These problems were solved with the P-EFC3 algorithm on a SUN Sparc 2, using the largest inconsistency mean for variable selection and the lowest inconsistency counts for value selection. We will refer to this version as LM (Largest Mean). Alternatively, we solved the problems using the highest weight heuristic (HW) and the lowest support heuristic (LS). To measure the performance of the different heuristics, we use the number of consistency tests and the CPU time required in the P-EFC3 execution. The number of

tests has been widely used as a measure for the computational effort needed for the algorithms to solve MAX-CSP instances. The CPU time is included to check the cost-effectiveness of the extra computation required by HW and LS with respect to the savings they produce in searching. These results are given in Table 1.

It can be observed that HW outperforms LM in all classes of problems, and this improvement increases with the problem difficulty, as p_1 and p_2 increases. The heuristic HW decreases drastically the computational effort needed to solve the problems. Globally, HW gives about a 60% saving in the number of tests with respect to LM. Regarding CPU time, HW clearly outperforms LM, despite the extra computation required to calculate weights. The performance of LS needs some more attention: for connectivities different from 1, the experiments show that LS is between HW and LM regarding both number of tests and CPU time (although closer to HW). However, for full connectivity its performance is similar to LM, with larger CPU times. As we will see in section 7, LS is approaching LM as problem connectivity tends to 1.

		LM		HW		LS	
p_1	p_2	consistencytests	CPU time	consistencytests	CPU time	consistencytests	CPU time
0.6	0.5	14,591	0.13	2,121	0.07	4,072	0.09
	0.6	79,472	0.48	25,049	0.26	36,208	0.33
	0.7	283,185	1.57	77,227	0.67	105,257	0.82
	0.8	870,141	4.75	147,975	1.19	236,862	1.77
	0.9	3,797,692	20.48	554,941	4.20	801,161	5.71
0.8	0.5	111,042	0.69	42,687	0.41	47,815	0.43
	0.6	512,095	2.82	146,414	1.19	190,075	1.46
	0.7	1,019,514	5.70	265,255	2.09	441,612	3.32
	0.8	3,065,557	17.03	589,559	4.54	1,040,506	7.74
	0.9	7,951,244	44.37	1,299,684	9.82	2,244,457	16.72
1.0	0.5	351,871	1.94	167,765	1.34	200,840	1.64
	0.6	973,165	5.33	439,824	3.62	903,770	7.03
	0.7	2,155,926	11.89	903,483	7.31	2,011,695	15.73
	0.8	5,783,030	32.64	2,155,868	17.31	5,053,403	38.90
	0.9	19,247,830	111.67	6,373,465	50.46	15,943,989	124.35

Table 1. Results on fixed tightness random CSP.

6.2 Variable Tightness Random CSP

We suspected that fixed tightness random CSP may be inappropriate to model some real CSP. Therefore, we tested the heuristics with variable tightness random CSP. The values for connectivity were again 0.6, 0.8 and 1. Three widths for the tightness interval were tried: for width = 0.2, we considered the intervals [0.4, 0.6], [0.6, 0.8] and [0.8, 1.0]. For width = 0.4, we considered the intervals [0.2, 0.6] and [0.6, 1.0]. Finally, for width = 1.0, the tightness interval was [0.0, 1.0]. Obviously, the larger the tightness interval is, the more heterogeneous is the problem sample. In order to produce representative results, we increased the number of instances considered for each parameter setting. Thus, for width of 0.2, 0.4 and 1, the number of instances was 40, 60 and 100, respectively, solving 1020 problem instances. Almost all these problems were unsolvable. Again, we report the number of consistency tests and CPU time for the heuristics LM, HW and LS. These results are in Table 2 for width = 0.2, in Table 3 for width = 0.4, and in Table 4 for width = 1.

Results show that HW performs similarly to the fixed tightness case with respect to LM in both number of consistency tests and CPU time. Considering LS, it can be seen a changing behaviour with increasing tightness width. For tightness width = 0.2, LS outperforms HW in problems with low connectivity and tightness, while HW outperforms LS in problems with high connectivity and tightness. For tightness width = 0.4, this behaviour is again observed, but HW can only surpass LS in problems with full connectivity and high tightness. For tightness width = 1, LS

		LM		HW		LS	
p_1	$p_2^{inf}\ p_2^{sup}$	consistency tests	CPU time	consistency tests	CPU time	consistency tests	CPU time
0.6	0.4, 0.6	28,155	0.12	4,378	0.09	3,542	0.08
	0.6, 0.8	394,793	2.10	78,766	0.66	73,881	0.57
	0.8, 1.0	7,657,250	41.23	931,949	7.04	739,405	5.12
0.8	0.4, 0.6	128,683	0.75	42,267	0.42	40,224	0.36
	0.6, 0.8	1,126,827	6.02	273,791	2.21	270,861	2.02
	0.8, 1.0	14,736,931	81.70	2,360,949	18.08	2,703,723	19.42
1.0	0.4, 0.6	431,766	2.32	185,132	1.54	174,265	1.35
	0.6, 0.8	2,337,982	12.69	851,510	6.75	1,114,632	8.27
	0.8, 1.0	31,427,200	181.20	8,488,424	67.27	15,318,073	116.41

Table 2. Results on variable tightness random CSP for width = 0.2.

		LM		HW		LS	
p_1	$p_2^{inf}\ p_2^{sup}$	consistency tests	CPU time	consistency tests	CPU time	consistency tests	CPU time
0.6	0.2, 0.6	2,832	0.05	634	0.06	606	0.06
	0.6, 1.0	2,565,439	13.70	328,493	2.50	237,123	1.70
0.8	0.2, 0.6	21,078	0.16	5,223	0.10	4,562	0.09
	0.6, 1.0	5,750,927	31.44	1,002,846	7.76	1,038,177	7.60
1.0	0.2, 0.6	101,440	0.62	44,307	0.44	38,178	0.37
	0.6, 1.0	9,897,304	55.76	2,773,760	21.78	4,019,892	30.07

Table 3. Results on variable tightness random CSP for width = 0.4.

		LM		HW		LS	
p_1	$p_2^{inf}\ p_2^{sup}$	consistency tests	CPU time	consistency tests	CPU time	consistency tests	CPU time
0.6	0.0, 1.0	688,492	3.98	78,666	0.71	42,969	0.34
0.8	0.0, 1.0	1,477,590	8.12	203,502	1.62	141,404	1.06
1.0	0.0, 1.0	1,629,693	8.77	306,932	2.46	255,381	1.93

Table 4. Results on variable tightness random CSP for width = 1.

outperforms HW in all problems classes. Therefore, LS seems to be a good heuristic for MAX-CSP problems with a high degree of variability on constraint tightness. For a medium degree of variability on constraint tightness, LS works better than HW for low tightness; for high tightness HW surpasses LS in fully connected problems only, showing a similar performance in the rest of the problems.

7 Discussion

Experimental results show that both heuristics, HW and LS, require less consistency tests than LM when used inside the P-EFC3 algorithm, that is, HW and LS are more informed than LM. The improvement that HW provides with respect to LM is approximately homogeneous in all problem classes where it was tested. On the other hand, the improvement that LS provides with respect to LM depends on the

connectivity and tightness interval. To explain these facts, we will show that LS follows the same lines than LM, but it goes one step further. Essentially, LS considers more information —the influence of future variables— and, because of that, it is able to guide better the search for a maximal consistent solution. In addition, we will explore the relation between LS and HW.

To analyze the LS heuristic, let us assume first that future variables are not considered in the gradient computation (i.e. $q_i^F[\lambda] = 0$ for all λ). Then, a gradient component for the variable X_i is,

$$q_i[\lambda_k] = q_i^P[\lambda_k] = 2(\#\text{past variables supporting } \lambda_k - \#\text{past variables not supporting } \lambda_k) =$$

$$= 2\,(card(\boldsymbol{P}) - 2\,inconsistency\_count\,(X_i, \lambda_k))$$

The sum of gradient components is,

$$\sum_{k=1}^{m} q_i[\lambda_k] = 2\,\{\, m*card(\boldsymbol{P}) \; - 2 \sum_{k=1}^{m} inconsistency\_count\,(X_i, \lambda_k) \;\}$$

It is easy to see that the variable which minimizes this expression is the variable with the largest mean of inconsistency counts, providing that no values are discarded for any variable by the P-EFC3 algorithm. The same thing applies for value ordering: considering values by increasing number of inconsistency count is equal to order them by decreasing gradient component. Therefore, disregarding the effect of future variables and providing that no values are discarded, LS is equivalent to LM.

If future variables are taken into account, the future gradient component $q_i^F[\lambda_k]$ is,

$$q_i^F[\lambda_k] = 2/m\,\{\, \sum_{j \in F} \sum_{l=1}^{m} r_{ij}(\lambda_k, \lambda_l) \;\} =$$

$$= 2/m\,\{\#\text{future variables with a value consistent with } \lambda_k - \#\text{future variables with a value inconsistent with } \lambda_k\,\}$$

The smaller $q_i^F[\lambda_k]$ is, the less consistent appears λ_k with respect to future variables. The sum of future gradient components provides an estimation of the consistency of the variable values with respect to future variables. Considering both past and future components, the sum of gradient components is,

$$\sum_{k=1}^{m} q_i[\lambda_k] = \sum_{k=1}^{m} q_i^P[\lambda_k] + \sum_{k=1}^{m} q_i^F[\lambda_k]$$

The variable with the lowest sum of gradient components will combine a high

number of inconsistencies with past variables with a high estimation of inconsistencies with future variables. In this sense, LS can be seen as a generalization of LM, including extra information from future variables.

According with experimental results, LS improvement with respect to LM is varying with tightness width. This can be explained from the following fact: for fixed tightness random CSP, the contribution of a future variable X_j in the sum of gradient components of any other future variable X_i with which it is constrained is approximately the same. That contribution is,

$$2/m \sum_{k=1}^{m} \sum_{l=1}^{m} r_{ij}(\lambda_k,\lambda_l) \approx 2m - 2mp_2$$

where $2m - 2mp_2$ is the expected value of the contribution of X_j. When connectivity is 1, all future variables are connected with each other, and the sum of their contributions on X_i is,

$$\sum_{k=1}^{m} q_i^F[\lambda_k] = 2/m \sum_{j\in F} \sum_{k=1}^{m} \sum_{l=1}^{m} r_{ij}(\lambda_k,\lambda_l) \approx (card(\boldsymbol{F}) -1) * (2m - 2mp_2) \quad (11)$$

where $(card(\boldsymbol{F}) -1) * (2m - 2mp_2)$ is the expected value of the sum of the contributions of future variables. This expected value is the same for each future variable X_i. For fixed tightness fully connected random CSP, the influence of future variables comes from deviations from (11), and typically it is small compared with the influence of past variables. This can be observed in Table 1, where LS approaches LM for $p_1 = 1$. For variable tightness random CSP, (11) is no longer the same for each future variable because constraints do not have a fixed tightness p_2. The wider the tightness interval is, the more informative is the influence of future variables, and the better is LS performance. This can be observed in Tables 2, 3 and 4.

The relation between LS and HW can be seen from the weight updating rule (6). LS selects the variable with the lowest denominator in (6). Typically, this variable will have some values with high weights, although not necessarily the highest. Therefore, in many cases LS will select a variable with high weights associated with values in its domains. These variable-value pairs stand for directions in which $A(P)$ increases. In addition, HW tends to select variables with a high variability in their gradient components, that is, variables in which some values appear as more promising than others (promising uphill directions of $A(P)$).

In summary, LS heuristic appears as a generalization of LM. It is based on the same insights, but it goes one step further by considering the influence that future variables have on selecting the current one. Thus, LS selects the variable with the lowest global support which is a combination of the supports of past and future variables. Regarding HW, it selects a variable with a value where $A(P)$ locally increases. Both LS and HW include information from past and future variables, in

contrast with LM which only considers information from past variables. Interestingly, the influence of past and future variables on LS and HW is proportional to their relative number. When no variables have been yet assigned, all the influence comes from future variables; when almost every variable has been assigned, the influence of future variables is very small. It is specially interesting the initial state, when no variable is assigned. LM decides randomly because it has no information (every inconsistency count is zero). This may lead to wrong decisions at early stages of the search, which are critical for search performance.

8. Conclusions

From this work, we extract the following conclusions. First, we have shown that our optimization formulation for CSP is a fruitful approach able to generate new heuristics. These heuristics are not simply "rules of thumb", but they are based on results about the best possible solution for a CSP. This substantiates these new heuristics, which appear to be better justified than other heuristics whose only justification comes from empirical results. Second, our optimization formulation is general (it does not differentiate between solvable and overconstrained CSP), and the heuristics based on it seems to be applicable to both problem types (although more empirical results are needed to qualify precisely this assertion). Third, the proposed approach gives a new perspective of previously applied heuristics, and it can be seen as a step further in the sense that it considers more information and, because of that, it provides better advice. Finally, we notice the practical importance of the incremental approach taken to compute weights in order to make these heuristics competitive against others of simpler computation.

Acknowledgements

We thank Carme Torras for several discussions about the topics of this paper. We thank Richard Cropper for his support in ironing the English. We also thank M. Luisa Romanillos and Romero Donlo for their collaboration on writing this paper.

References

Bakker R., Dikker F., Tempelman F. and Wognum P. (1993). Diagnosing and solving overdetermined constraint satisfaction problems, *Proceedings of IJCAI-93*, 276-281.

Feldman R. and Golumbic M. C. (1990). Optimization algorithms for student scheduling via constraint satisfiability, *Computer Journal*, vol. 33, 356-364.

Fox M. (1987). *Constraint-directed Search: A Case Study on Jop-Shop Scheduling*. Morgan-Kauffman.

Freuder E. C. and Wallace R. J. (1992). Partial constraint satisfaction, *Artificial Intelligence*, 58: 21-70.

Horst R. and Tuy H. (1993). *Global Optimization (2 edition)*, Springer-Verlag.

Hummel R. A. and Zucker S. W. (1983). On the Foundations of Relaxation Labeling Processes, *IEEE Trans. Pattern Analysis Machine Intelligence*, vol. 5, no. 3, 267-287.

Meseguer P. and Larrosa J. (1995). Constraint Satisfaction as Global Optimization, *Proceeedings of IJCAI-95*, in press.

Prosser P. (1994). Binary constraint satisfaction problems: some are harder than others, *Proceedings of ECAI-94*, 95-99.

Rosenfeld A., Hummel R. and Zucker S. (1976). Scene Labeling by Relaxation Operators, *IEEE Trans. Systems, Man, Cybernetics*, vol. 6, no.6, 420-433.

Sastry *P*. S. and Thathachar M. A. L. (1994). Analysis of Stochastic Automata Algorithm for Relaxation Labeling, *IEEE Trans. Pattern Analysis Machine Intelligence*, vol. 16, no. 5, 538-543.

Wallace R. J. and Freuder E. C. (1993). Conjunctive width heuristics for maximal constraint satisfaction, *Proceedings of AAAI-93*, 762-778.

Wallace R. J. (1994).Directed Arc Consistency Preprocessing as a Strategy for Maximal Constraint Satisfaction, *ECAI94 Workshop on Constraint Processing*, M. Meyer editor, 69-77.