Neural Comput & Applic (2010) 19:1133–1142
DOI 10.1007/s00521-010-0390-8

AIS

A graph-based immune-inspired constraint satisfaction search

María-Cristina Riff · Marcos Zúñiga · Elizabeth Montero

Received: 31 August 2007 / Accepted: 19 November 2008 / Published online: 30 June 2010

Abstract We propose an artificial immune algorithm to solve constraint satisfaction problems (CSPs). Recently, bio-inspired algorithms have been proposed to solve CSPs. They have shown to be efficient in solving hard problem instances. Given that recent publications indicate that immune-inspired algorithms offer advantages to solve complex problems, our main goal is to propose an efficient immune algorithm which can solve CSPs. We have calibrated our algorithm using relevance estimation and value calibration (REVAC), which is a new technique recently introduced to find the parameter values for evolutionary algorithms. The tests were carried out using randomly generated binary constraint satisfaction problems and instances of the three-colouring problem with different constraint networks. The results suggest that the technique may be successfully applied to solve CSPs.

1 Introduction

Constraint satisfaction problems (CSPs) involve finding values for problem variables, subject to constraints which must be satisfied. Many real-world tasks can be expressed as CSPs. Some well-known CSPs include the graph k-colouring problem, the satisfiability problem, the scene-labelling problem, temporal reasoning, resource allocation, planning, scheduling and graph matching. Over the past few years, many algorithms and heuristics have been developed to solve CSPs. Following trends from the constraint research community, the bio-inspired computation community has proposed some other approaches to tackle CSPs, such as evolutionary algorithms [2, 6, 7, 10, 12, 16] and ants algorithms [15]. Given that recent publications indicate that artificial immune systems offer advantages in solving complex problems [3, 5], our main goal is to propose an efficient immune-inspired algorithm which can solve CSPs. We restrict our attention to binary CSPs, where the constraints involve two variables. If a variable i has a domain of potential values D_i and a variable j has a domain of potential values D_j, the constraint on i and j, R_{ij}, is a subset of the Cartesian product of D_i and D_j. A pair of values (a, b) is called consistent, if (a, b) satisfies the constraint R_{ij} between i and j. A binary CSP is associated with a constraint graph, where nodes represent variables and arcs represent constraints. Artificial immune systems, as well as evolutionary algorithms, are very sensitive to their parameter values. Garrett in [8] proposed a parameter-free clonal selection using adaptive changes. In this paper, we use a recently proposed method for tuning [11]. This method uses statistical properties to determine the best set of parameter values for an evolutionary algorithm. The contributions of this paper are the following:

- An immune-inspired algorithm guided by the constraints graph which can solve hard CSPs,
- A new application of the tuning method relevance estimation and value calibration (REVAC) proposed for evolutionary algorithms [11].

Marcos Zúñiga has been financed as Scientific Assistant DGIP, UTFSM.

Supported by Fondecyt Project 1080110.

M.-C. Riff (✉) · M. Zúñiga · E. Montero
Department of Computer Science,
Technical University Federico Santa María,
Valparaíso, Chile
e-mail: mcriff@inf.utfsm.cl

M. Zúñiga
e-mail: mzuniga@inf.utfsm.cl

E. Montero
e-mail: emontero@inf.utfsm.cl

The first steps of this research have been presented in [13]. This paper is structured as follows. In the next section, we describe the framework to be used to design our approach. In Sect. 3, we define the binary constraint satisfaction problem. In Sect. 4, we introduce our new approach: CD-NAIS. The results of tests and a comparison with other incomplete methods are presented in Sect. 5. In the last sections, we present conclusions and future work.

2 Artificial immune systems

Artificial immune systems (AIS) are adaptive systems inspired by immunological theory [3]. From the information processing point of view, an AIS is a parallel and a distributed adaptive system. It uses learning, memory and associated recovery to do recognition and classification tasks. Roughly speaking, the principal function of an immune system is to protect the individual from the repeated attacks of external agents. The system recognises and discards doing an immune answer coming from one of the two levels: the innate immune system or the adaptive immune system. The cells of the innate immune system are immediately able to take action against external attacks. In the adaptive immune level, the antibodies are produced as an answer to specific infections. These cells can develop a memory, thus they will be able to recognise a future similar attack. L. de Castro [3] proposed a framework to design an AIS with the following components:

1. A representation to create abstract models of organs, cells and immune molecules (antigen, antibody). A molecule can, in general, be represented by a shape space S like an attribute chain (set of coordinates) of size L. Thus, an attribute chain $m = < m_1, m_2, ..., m_L >$ corresponds to a point in the shape space.
2. A set of functions called affinity functions to quantify the interactions between the AIS components (organs, cells and molecules).
3. A set of algorithms to simulate the immune behaviour. In this work, we use two of the most known models: the clonal selection and the immune network. The clonal selection manages the interaction of the immune system components. The immune network is used to simulate both the dynamic and meta-dynamic behaviour.

This framework is a general guide used to design artificial immune systems. When using this framework to design an immune algorithm for a specific problem, the main research task consists of defining all of its components, the same way that this is done for other techniques like genetic algorithms. To help the immune algorithm's search, this design process must take into account the knowledge of the problem, and as far as possible, some knowledge coming from other techniques. The effort of the constraints research community has been traditionally focused on developing methods to improve the performance of the algorithm using the knowledge of the constraints, like pruning the search space. In order to use the knowledge about constraints in immune algorithms, we focus on the constraints graph that is defined in the following section.

3 Binary constraint satisfaction problems

We consider a CSP as defined by Mackworth [9], which can be stated briefly as follows: given a set of variables, a domain of possible values for each variable, and a conjunction of constraints, each constraint is a relation defined over a subset of the variables, limiting the combination of values that the variables in this subset can take. The goal is to find a consistent assignment of values to the variables so that all the constraints are satisfied simultaneously. CSPs are, in general, NP-complete problems and some are NP-hard [1]. Thus, a general algorithm designed to solve any CSP will, in the worst case, require exponential time in problem solving. A CSP is composed of a set of *variables* $V = \{X_1, ..., X_n\}$, their related *domain set* $D = \{D_1, ..., D_n\}$ and a set θ containing η *constraints* on these variables. The domain of a variable is a set of values to which the variable may be instantiated. Each variable X_j is *relevant* to a subset of constraints $C_{j\_1}, ..., C_{j\_k}$, where $\{j_1, ..., j_k\}$ is some subsequence of $\{1, 2, ..., \eta\}$. A constraint which has exactly one relevant variable is called a *unary constraint*. Similarly, a *binary constraint* has exactly two relevant variables. If two values assigned to variables that share a constraint are not among the acceptable value-pairs of that constraint, this is an *inconsistency* or constraint violation. We can represent a binary CSP by a constraint graph defined as follows:

Definition 1 (Constraint Graph) The constraint graph of a CSP (V, D, θ) is a graph, in which each node represents a variable in V, and each edge represents a constraint in θ.

4 CD-NAIS: a constraint-directed network artificial immune system

We have called our algorithm CD-NAIS which stands for constraint-directed network artificial immune system. The algorithm uses three immune components: antigen, antibody, and B-cells. Basically, the antigen represents the information for each variable given by the constraint graph. This information is related to the number of connections of

each variable, that is, the number of constraints where each variable is a *relevant variable*. Thus, it is fixed information and does not depend on the state of the search of the algorithm. On the contrary, the antibody does strongly depend on the state of the search of the algorithm. It has two kinds of information: the variable values and the constraints violated under this instantiation. Lastly, a B-cell has all the antibody information required by the algorithm for its evolution.

4.1 Immune components for a CSP

The immune components in our approach are defined as follows:

Definition 2 (Antigen) For a CSP and its constraint graph, we define the antigen Ag of the n-tuple of variables $(\mathrm{Ag}_1, ..., \mathrm{Ag}_n)$, such that the Ag_i value is the number of constraints where X_i is a relevant variable, $\forall i$, $i = 1, ..., n$.

Thus, the antigen represents the maximal number of inconsistencies for each variable. The algorithm needs to know, for each pre-solution, its variable values and the constraints satisfied under this instantiation. For this reason, the antibody has two segments: a structural and a conflicting segment.

Definition 3 (Structural Antibody) A *Structural Antibody* Ab_s is a mapping from a n-tuple of variables $(X_1, ..., X_n) \rightarrow D_1 \times \cdots \times D_n$, such that it assigns to each variable in V a value from its domain. The structural antibody corresponds to an instantiation of the CSP.

Definition 4 (Conflicting Antibody) For a CSP and its constraints graph we define the *Conflicting Antibody* Ab_c of the n-tuple of variables $(\mathrm{Ab}_{c1}, ..., \mathrm{Ab}_{cn})$, such that the Ab_{ci} value is the number of *violated* constraints, where X_i is a relevant variable, $\forall i$, $i = 1, ..., n$.

A solution consists of a structural antibody which does not violate any constraint, that is, whose conflicting antibody complements the antigen. Before defining the B-cell, we need to introduce the idea of affinity in the context of our problem.

4.2 Affinity measure

For artificial immune systems, affinity is a measurement of the interaction between two immune components. In our approach, we are interested in two kinds of affinity. The affinity between the antigen and a conflicting antibody, and the affinity between two structural antibodies.

- Interaction between $\mathrm{Ag} \leftrightarrow \mathrm{Ab}_c$:
  It is an estimation of how far the antibody is from being a CSP solution. The key idea is that a solution of the CSP corresponds to the largest value of the affinity function between Ab_c and Ag. This occurs when all the constraints are satisfied. We define the function A_{csp} to measure this affinity as:

$$A_{\mathrm{csp}}(\mathrm{Ag}, \mathrm{Ab}_c) = \sqrt{\sum_{i=1}^{n} (\mathrm{Ag}_i - \mathrm{Ab}_{c_i} + \mathrm{Fd}_i)^2} \quad (1)$$

where n is the number of variables of the CSP problem, Fd_i is called the dispersion factor defined by:

$$\mathrm{Fd}_i = \frac{d_i}{(n-1) \times \mathrm{Ag}_i} \quad (2)$$

with d_i equal to:

$$d_i = \sum_{i \neq j, j=1}^{n} |\mathrm{Ab}_{c_i} - \mathrm{Ab}_{c_j}| \quad (3)$$

The function A_{csp} does not only prefer a pre-solution with a minimal number of violated constraints, but it also takes into account how hard for the algorithm to repair this pre-solution could be. This is done by including the function Fd_i as a conflict dispersion measure. Thus, given two pre-solutions which satisfy the same number of constraints, the algorithm prefers the one with the smallest number of variables (nodes) involved in the constraints violations. The value of the dispersion factor Fd_i belongs to [0, 1). The Fd_i value is equal to 0, either when any of the variables are in conflict (it is a solution) or when all the variables are involved in the same number of conflicts. The following examples illustrate how these functions are used. Consider $n = 3$, $\mathrm{Ag} = \{5, 5, 5\}$, and two pre-solutions to compare, $\mathrm{Ab1} = \{5, 1, 5\}$ and $\mathrm{Ab2} = \{5, 2, 5\}$. They differ in one conflict less for Ab1 compared to Ab2, in variable 2. For Ab1, $\mathrm{Fd}_1 = 0.4$, $\mathrm{Fd}_2 = 0.8$, $\mathrm{Fd}_3 = 0.4$, giving as result $A_{\mathrm{csp}} = 2.227$. For Ab2, $\mathrm{Fd}_1 = 0.3$, $\mathrm{Fd}_2 = 0.6$, $\mathrm{Fd}_3 = 0.3$, giving as result $A_{\mathrm{csp}} = 1.881$. Then, Ab1 is preferred, demonstrating that, even with just one conflict of difference, the number of conflicts is mandatory over the dispersion factor to determine the best pre-solution.

Now consider the example in Fig. 1. It is a three-colouring problem with five variables whose domains are the three colours red (R), yellow (Y) and green (G) and the constraints are that two connected nodes cannot have the same colour. Given the instantiations $\mathrm{Ab}_{s1} = \{R, Y, R, G, G\}$ and $\mathrm{Ab}_{s2} = \{R, Y, Y, Y, R\}$ both having

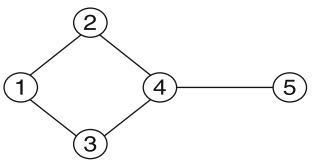

Fig. 1 Example: three-colouring problem

two conflicts, the function used in our algorithm prefers the second instantiation. This is because there are just three variables involved in the constraints violation (2, 3, 4) in the second instantiation, instead of the four variables (1, 3, 4, 5) involved in the first instantiation. The dispersion factor allows the algorithm to discriminate between these instantiations. The affinity values are $A_{\text{csp}} = 3.17$ for Ab_{s1} and $A_{\text{csp}} = 3.82$ for Ab_{s2}.

- Interaction between $\text{Ab}_{si} \leftrightarrow \text{Ab}_{sj}$:
  The idea of using this measure, called $\text{HA}_s(\text{Ab}_{si}, \text{Ab}_{sj})$, is to quantify how similar two pre-solutions are. To compute this interaction, our algorithm uses the Hamming distance, normalised by the number of variables n. The algorithm prefers to have a diversity of pre-solutions. This affinity measure will allow the antibody network to keep the diversity of its elements.

4.3 B-cell representation

A B-cell is a structure with the following components:

- An antibody $\text{Ab} = (\text{Ab}_c, \text{Ab}_s)$.
- The number of clones of Ab to be generated for the clonal expansion procedure. This number is directly proportional to the A_{csp} value.
- The hypermutation ratio used in the affinity maturation step. This ratio is inversely proportional to the A_{csp} value.

4.4 The algorithm: constraint-directed network artificial immune system

The CD-NAIS algorithm is shown in Fig. 2. The algorithm works with a set of B-cells, following an iterative maturation process. First, the *Antigen Presentation* procedure calculates the affinity $Ag \leftrightarrow \text{Ab}_c$. The returned affinity value corresponds to the result of $A_{\text{csp}}(\text{Ag}, \text{Ab}_c)$, presented in Eq. 1. Some of these B-cells are selected performing a *Clonal Selection*, which prefers those B-cells with higher affinity values A_{csp}, considering the B-cells selection rate n_1. It uses a Roulette Wheel selection procedure. The algorithm generates clones of the B-cells selected. This is done by the *Clonal Expansion* procedure, where a fixed size population of clones is generated proportionally to the A_{csp} affinity of the selected B-cells (for more details, see Sect. 4.5.1). These clones follow a hypermutation process in the *Affinity Maturation* step, which performs a local search procedure that tries to repair the structural antibody Ab_s. It is explained in Sect. 4.5.2. The new set of B-cells is composed of a selected set of hypermutated clones. This selection is done in the *Build Network* procedure, using the Hamming Distance HA_s between the structural antibodies of the hypermutated clones in order to have a diversity of new B-cells. The best antigen affinity hypermutated clones will have priority to be included in the set of new B-cells. This procedure is detailed in Sect. 4.5.3. The algorithm adds new B-cells randomly generated by the *Meta-dynamic* procedure to this set of B-cells, suppressing the lower affinity B-cells, by a pre-defined new B-cells insertion rate n_2. Thus, the algorithm does exploration and exploitation. This process is repeated until a pre-defined number of iterations or until a solution is found.

4.5 Other immune processes involved with CD-NAIS

Several CD-NAIS procedures involving important immune processes will be reviewed in more detail. The following sections will present these procedures.

Fig. 2 CD-NAIS pseudocode

```
Algorithm CD-NAIS(CSP) returns memory B-cells
Begin
    Ag ← Determine constraint graph connections(CSP, n);
    Initialise B-cells
    For i ← 1 to bcells_number do                          (1) Antigen Presentation
        Compute affinity value A_csp(B-cells[i])
    End For
    j ← 1;
    While (j ≤ MAX_ITER) or (not solution) do
        Select a set of B-cells by Roulette Wheel          (2) Clonal Selection
        Generate Clones of the selected B-cells            (3) Clonal Expansion
        Hypermutate Clones                                 (4) Affinity Maturation
        For k ← 1 to CLONES_NUM do
            Compute affinity value A_csp(Clones[k])
        End For
        B-cells ← build_network(CLONES);
        B-cells ← metadynamics(B-cells);
    End While
    Return B-cells;
End
```

4.5.1 Clonal expansion

The *clonal expansion* procedure, presented in Fig. 3, generates a set of clones using the set of B-cells, previously selected by the *clonal selection* procedure. The constant parameter *clones_number* represents the total number of clones to be generated. The hypermutation rate of a B-cell belongs to the interval [*min_rate*, *max_rate*], and it is inversely proportional to the affinity of the B-cell. This procedure generates the clones directly proportionally to the affinity of each B-cell, in order to generate a fixed size set of *clones_number* clones. This procedure also sets the hypermutation rate for each clone as the hypermutation rate of the spawning B-cell, used by the *Affinity Maturation*, which is detailed in the following section.

4.5.2 Affinity maturation

The affinity maturation procedure, presented in Fig. 4, performs a hypermutation process to a set of *CLONES*, modifying the variable values represented in the structural antibody Ab_s. This procedure uses the information given by the conflicting antibodies Ab_c, guiding the hypermutation to the variables involved in more conflicts. Given a clone, the algorithm computes the number of conflicts where each variable is involved. This value is normalised according to the variable which is involved in the highest number of conflicts. The value obtained is called *conflicts_rate* and belongs to the interval [0, 1]. A weighted rate *mod_rate* is computed using both the hypermutation rate of the clone and *conflicts_rate*, to guide the hypermutation procedure to

Fig. 3 Pseudo-code for *clonal expansion* procedure, used by CD-NAIS algorithm

```
procedure Clonal Expansion (SELECTED_BCELLS)
Begin
selected ← bcells_number × (1 − n1);
total_clones ← 0;
For i ← 1 to selected do
    clones_per_bcell ← clones_number × SELECTED_BCELLS[i].affinity;
    h_rate ← (min_rate − max_rate) × SELECTED_BCELLS[i].affinity + max_rate;
    For j ← 1 to clones_per_bcell do
        CLONES[total_clones] ← SELECTED_BCELLS[i];
        CLONES[total_clones].hypermutation ← h_rate;
    total_clones ← total_clones + 1;
    End For
End For
Return CLONES;
End
```

Fig. 4 Pseudo-code for *affinity maturation* procedure, used by CD-NAIS

```
procedure Affinity Maturation (CLONES)
Begin
For i ← 1 to clones_number do
    max_conflicts ← 0;
    For j ← 1 to n do
        If CLONES[i].Abc[j] > max_conflicts then
          max_conflicts ← CLONES[i].Abc[j];
        End If
    End For
    For j ← 1 to n do
        conflicts_rate ← CLONES[i].Abc[j]/max_conflicts;
        rvalue ← Random (0, 1);
        mod_rate ← (α × conflicts_rate + β × CLONES[i].hypermutation)/(α + β);
        If rvalue < mod_rate then
          variable_value ← m× Random (0, 1);
          If variable_value improves CLONES[i]
            Or Random (0, 1) < exploration_rate then
            CLONES[i].Abs[j] ← variable_value;
          End If
        End If
    End For
End For
Return CLONES;
End
```

variables that are involved in a greater number of conflicts. In Fig. 4, constants α and β represent the weights for the *conflicts_rate* and hypermutation rate, respectively. The computed weighted rate *mod_rate* is used to randomly determine whether a variable will be mutated or not. If a variable is selected to be mutated, the new value for a variable is randomly chosen from its domain. The variable value is changed when the new chosen value reduces the number of conflicts of the clone. If this new value does not improve the clone affinity, it can still be selected according to a predefined probability *exploration_rate*. Thus, this procedure performs exploration and exploitation of the search space. Exploitation is performed by the hypermutation which is focused on repairing the variables involved in more conflicts. For exploration, the algorithm allows acceptance of pre-solutions with worse affinity, instead of the current one.

4.5.3 Immune network construction

The *build network* procedure uses the immune network concepts to generate a set of diverse memory B-cells. This procedure, described in Fig. 5, uses a set of hypermutated clones ordered from the highest to the lowest affinity A_{csp} to generate a set of memory B-cells.

Fig. 5 Pseudo-code for *build network* procedure, used by CD-NAIS

```
procedure Build Network (CLONES, n)
Begin
For i ← 1 to clones_number do
    added_clone[i] ← false;
    suppression_counters[i] ← 0;
End For
num_added_b_cells ← 0;
b_cells_to_be_selected ← bcells_number × (1 − n2);
current_clone_index = 0;
current_clone ← CLONES[current_clone_index];
While (num_added_b_cells < b_cells_to_be_selected) do
    MEMORY_BCELLS[num_added_b_cells] ← current;
    num_added_b_cells ← num_added_b_cells + 1;
    added_clone[current_clone_index] ← true;
    For i ← 1 to clones_number do
        If added_clone[i] = false then
            network_affinity ← HA_s(current_clone.Ab_s, CLONES[i].Ab_s);
            If network_affinity < ε then
                suppression_counters[i] ← suppression_counters[i] + 1;
            End If
        End If
    End For
    clone_found ← false;
    For i ← current_clone_index to clones_number do
        If suppression_counters[i] = 0 and not then
            current_clone_index ← i;
            current_clone ← CLONES[current_clone_index];
            clone_found ← true;
            Break For;
        End If
    End For
    If clone_found = false then
        Break While;
    End If
End While
For i ← num_added_b_cells to b_cells_to_be_selected do
    least_suppressed ← clones_number;
    For j ← 0 to clones_number do
        If added_clone[j] = false and suppression_counters[j] < least_suppressed then
            least_suppressed ← suppression_counters[j];
            current_clone_index ← j;
        End If
    End For
    MEMORY_BCELLS[i] ← CLONES[current_clone_index];
    added_clone[current_clone_index] ← true;
End For
Return MEMORY_BCELLS;
End
```

A suppression counter is defined in order to evaluate how similar a clone is to the clones in memory. The key idea is the diversity. The selection process is done by including clones that are quite different from the clones that are already in memory. It begins by the clones with a suppression counter equal to zero. At each step, the suppression counter is re-computed considering the new clone included in memory. The suppression counter of the clones that are similar to the new one in memory will be incremented if $HA_s(Ab_{si}, Ab_{sj}) < \epsilon$, where i is the clone currently added to the memory B-cells set. Therefore, the ϵ value is used by the algorithm to manage the minimal degree of diversity. The ϵ value is known as the *threshold of crossing reactivity*. When no more clones with a suppression counter equal to zero are available, the clones with the lowest suppression counters are incorporated, until the number of B-cells in memory has been completed.

5 Experimental results

We have used a collection of two different types of problems to evaluate the performance of our algorithm. The first type of problems is a set of randomly generated binary CSPs, [14], and the second one is a set of hard instances of the well-known three-colouring problem. We describe both types of problems in the following sections.

5.1 Random binary CSPs

For random binary CSPs, we compared CD-NAIS with GSA [6], which is a sophisticated evolutionary algorithm used to solve CSPs and strongly uses knowledge derived from the constraints research community. We also compared CD-NAIS with SAW [2]. SAW has been compared with both complete and incomplete well-known algorithms for CSPs obtaining better results in most of the cases tested. The idea of these tests was to study the behaviour of the algorithm when solving hard problems. We used two models to generate binary CSPs. That is because GSA has been reported using model B proposed in [14], and SAW has been reported using model E [2].

5.1.1 Model B

The binary CSPs belonging to the hard zone are randomly generated using the model proposed by B. Smith in [14]. This model considers four parameters to obtain a CSP. That is, the number of variables (n), the domain size for each variable (m), the probability p_1 that a constraint exists between two variables, and the probability p_2 of compatibility values. This model exactly determines the number of constraints and the number of consistent instantiations for the variables that are relevant for a given constraint. Thus, for each set of problems randomly generated, the number of constraints is $\frac{p_1 n(n-1)}{2}$, and for a given constraint the number of consistent instantiations is $m^2 p_2$. Given (n, m, p_1) B. Smith defines a function to compute critical p_2 values ($\hat{p}_{2_{\text{crit}}}$ values, defined in Eq. 4). These values allow obtaining CSPs in the transition phase, which correspond to problems that are harder to be solved.

$$\hat{p}_{2_{\text{crit}}}(n, m, p_1) = m^{-\frac{2}{(n-1)p_1}} \tag{4}$$

5.1.2 Model E

This model also considers four parameters (n, m, p, k). The parameters n and m have the same interpretation than in model B. For binary CSPs whose constraints have two relevant variables, $k = 2$ in model E. The p parameter corresponds to the probability that a pair of values for a pair of variables is valid. The higher p the more difficult, on average, the problem instances will be.

5.2 3-Colouring problem

The goal of this test is to evaluate the performance of CD-NAIS in front of real-world cases as the well-known three-colouring problem. Applications of the 3-colouring problem are scheduling, frequency assignment, or register allocation. We have used the Joe Culberson library[1] to generate random graphs to be coloured. These graphs have at least one solution.

5.3 Hardware

The hardware platform for the experiments was a PC Pentium IV Dual Core, 3.4Ghz with 512 MB RAM under the Linux Mandriva 2006 operating system. The algorithm has been implemented in C. The code for CD-NAIS is available on a website[2]. The tests allowed to evaluate the performance of CD-NAIS when it was calibrated using the REVAC technique for tuning. In the following section, we describe the tuning process for CD-NAIS.

5.4 REVAC

The relevance estimation and value calibration (REVAC) has recently been proposed in [11]. The goal of this algorithm is to determine the parameter values for evolutionary algorithms. It is also able to identify which parameters are not relevant for the algorithm. Roughly speaking, REVAC is a steady-state evolutionary algorithm for tuning another

[1] http://www.web.cs.ualberta.ca/joe/.

[2] http://www.sop.inria.fr/orion/personnel/Marcos.Zuniga/CSPsolver.zip

Fig. 6 Success rate and CPU time for CD-NAIS and GSA

Category	GSA		CD-NAIS					
	50,000 ev.	time [s]	10,000 ev.	time [s]	50,000 ev.	time [s]	75,000 ev.	time [s]
c0.3_t0.7	93.33	2.18	87.5	0.68	93.75	1.85	97.5	1.98
c0.5_t0.5	84.4	2.82	91	0.61	98.33	1.08	99.33	0.98
c0.5_t0.7	100	2.76	84	0.61	83.33	2.79	84.33	3.91
c0.7_t0.5	16.7	10.35	87	0.77	87	2.09	90.67	3.34
c0.7_t0.7	100	2.51	80.67	0.66	80.33	2.15	82	4.52
c0.9_t0.5	3.3	13.77	83.67	0.52	86.33	2.15	87.33	4.19
c0.9_t0.7	99.0	1.58	75.33	0.82	75.33	3.92	75.67	6.03

evolutionary algorithm. REVAC uses a real-value representation, where each value corresponds to a parameter value of the algorithm to be tuned. Each chromosome in REVAC is evaluated by the performance obtained by the algorithm (to be tuned) using its parameter values. It uses recombination and mutation operators. In order to apply REVAC to calibrate CD-NAIS, we have selected 14 problems, two from each category <10, 10, p_1, p_2> using model B. The performance for each chromosome is computed as the number of satisfied constraints by the solution obtained by CD-NAIS using the parameter values of the chromosome. The parameter values found by this tuning procedure were as follows:

- $n_1 = 0.3$, rate of cells to be expanded
- $n_2 = 0.4$, rate of cells to be incorporated on the memory
- $\epsilon = 0.40$, reactivity threshold between clones
- Number of B-cells = 5
- Number of clones = 100

This means that CD-NAIS requires doing more exploration than it does using a hand-made calibration. In the hand-made calibration, the hypermutated cell is accepted if it differs at least by 54% ($\epsilon = 0.46$) from the memory cells. Now, it must differ at least by 60% to be accepted. Furthermore, the number of cells to be expanded has been reduced by 0.2. The procedure required around 14 h, computational time, to determine these parameter values.

5.5 Comparison between CD-NAIS and GSA using model B

Because the reported results of GSA [6] have been evaluated with the problems in the hardest zone, as they were generated in [14], we ran the calibrated CD-NAIS using problems generated with the parameters (n, m, p_1, p_2). We considered the problems with $n = m$, where $n = 10$. The p_1 and p_2 values are those belonging to the hardest zone. The table in Fig. 6 shows the percentage of problems solved and the time required for GSA, and those obtained by CD-NAIS considering 10, 000, 50, 000 and 75, 000 evaluations. In this table, the category $c0.3\_t0.7$ means $p_1 = 0.3$ and $p_2 = 0.7$, and so on.

p	SAW		CD-NAIS					
	100,000 ev.		10,000 it.		50,000 it.		75,000 it.	
	success	time [s]	success	time [s]	success	time [s]	success	time [s]
0.24	100	0.74	100	0.32	100	0.33	100	0.3
0.25	100	2.33	100	0.44	100	0.43	100	0.4
0.26	97	6.83	100	0.56	100	0.6	100	0.61
0.27	60	11.39	100	1.2	100	1.1	100	1
0.28	25	18.66	98.4	2.06	100	2.26	100	2.05
0.29	17	20.57	84	4.11	99.6	5.24	100	5.41
0.3	5	22.27	47.6	6.99	84.4	17.17	90	21.02
0.31	1	22.47	16.8	8.62	38.4	35.1	46.8	48.91
0.32	0	22.39	24	8.22	59.6	27.64	63.6	37.95
0.33	1	22.38	24.4	8.3	59.6	27.4	68.4	34.85

Fig. 7 Success rate and CPU time for CD-NAIS and SAW

CD-NAIS has a higher satisfaction rate than GSA; moreover, it converges very quickly to good solutions. Furthermore, in considering just 10,000 evaluations, the average success rate for CD-NAIS was around 83% instead of 72% for GSA. However, in some categories, GSA outperforms CD-NAIS. In order to be more exact in our comparison, we wanted to also use REVAC for tuning GSA. However, the GSA code is not available and some designing considerations lack precise specification in the original publication.

5.6 Comparison between CD-NAIS and SAW using Model E

We have generated 250 problem instances in the hard zone using Model E. Figure 7 shows the success rate and the time in seconds for both algorithms. We can observe that CD-NAIS outperforms SAW in both time and success rate. Moreover, the average success rate for SAW is 40.6% instead of a 69.2% for CD-NAIS, in just considering 10,000 iterations. CD-NAIS has just required 4.1 s, on average, for this number of iterations. Figure 8 shows the results for CD-NAIS and SAW. We can observe the transition phase in $p = 0.31$ for CD-NAIS.

We have also tuned SAW using REVAC. The parameters found by REVAC are different from the reported in the original work. The results obtained by SAW using the parameters values found by REVAC are shown in Fig. 9. For most of the tested problems, we have obtained better results with SAW using hand-made tuning than using the parameter values found by REVAC. Therefore, REVAC

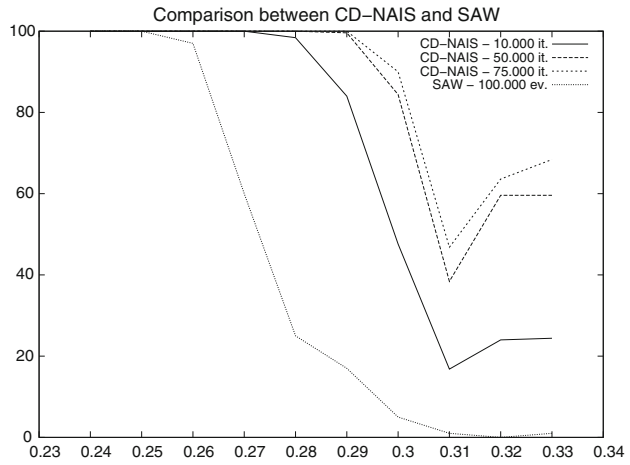

Fig. 8 Different problems tested, comparison of % successful runs

p	0.24	0.25	0.26	0.27	0.28	0.29	0.30	0.31	0.32	0.33
Success Rate	100	98.4	90.4	61.2	25.2	14.8	3.42	0.6	0	0.4

Fig. 9 SAW success rate tuned using REVAC

does not significantly improve the results obtained with SAW. In only one case ($p = 0.27$), REVAC has allowed an improvement of 1.2% of the SAW results.

5.7 Tests for 3-colouring problems

The idea of these tests is just to show that CD-NAIS can solve real-world cases. For these tests, we have generated graphs with a sparse topology, as they are known to be the most difficult ones to solve. For each number of constraints from {90, 120, 150, 180, 210, 240, 360}, we have generated 100 random 3-colouring graph problems. In order to discard the *easy problems*, we have applied DSATUR [4]. DSATUR is one of the best algorithms to solve this kind of problems. Finally, the problems used for testing are 50 problems, for each category, selected from those that DSATUR cannot solve. Figure 10 shows the percentage of graphs solved by CD-NAIS using 10,000 iterations. We have also included the percentage of graphs solved by CD-NAIS using other parameter configurations with just 10 B-cells and 3 clones found by tuning. It shows that the algorithm performance is sensitive to the parameter values.

Number of Constraints	% of Graphs Solved	% of Graphs Solved (B-cells, clones)=(10,3)
60	96.2	100
90	91.9	99.6
120	88.0	100
150	77.6	98.8
180	70.6	99.2
210	63.2	98.2
240	55.7	94.4
360	41.2	74.2

Fig. 10 Percentage of graphs solved for the three-colouring problem

CD-NAIS is able to solve some hard instances of the three-colouring problems. Furthermore, like for all other known solvers, the problems in the transition phase are also the most difficult ones for CD-NAIS.

6 Conclusions

Artificial immune systems have some interesting characteristics from the computational point of view: pattern recognition, affinity evaluation, immune networks, and diversity. All of these characteristics have been included in our algorithm. The B-cell structure is useful to determine both the solution of the problems and also to identify conflicts. The conflicting antibody is used by the algorithm to guide the reparation of the solutions (hypermutation process), giving more priority to the variables involved in a higher number of conflicts. For the problems in the hardest zone CD-NAIS solved, on average, 28% more problems than SAW (one of the best known evolutionary algorithms), just using 10,000 iterations (avg. 4.1 s). The calibrated CD-NAIS solved more problems than GSA, which is a sophisticated genetic algorithm that incorporates many constraint concepts to solve a CSP. Artificial immune systems are promising techniques to solve constrained combinatorial problems.

7 Future work

The use of parameter control strategies into the algorithm is a promising research area. This is mainly because the tuning process used to define the parameter values for CD-NAIS is a time-consuming task. Moreover, the process of changing the parameter values during the search has shown to be a key idea in helping evolutionary algorithms converge to find near-optimal solutions. We expect that CD-NAIS would benefit from using similar strategies.

References

1. Cheeseman P, Kanefsky B, Taylor W (1991) Where the really hard problems are. In: proceedings of the international joint conferences on artificial intelligence (IJCAI91), pp 163–169
2. Craenen BGW, Eiben AE, van Hemert JI (2003) Comparing evolutionary algorithms on binary constraint satisfaction problems. IEEE Trans Evol Comput 7(5):424–444
3. de Castro LN, Timmis J (eds) (2002) Artificial immune systems: a new computational intelligence approach. Springer, London, UK

4. Brelaz D (1979) New methods to color the vertices of a graph. Commun ACM 22:251–256
5. Dasgupta D. (eds) (2000) Artificial immune systems and their applications. Springer, Berlin
6. Dozier G, Bowen J, Homaifar A (1998) Solving constraint satisfaction problems using hybrid evolutionary search. IEEE Trans Evol Comput 2(1):23–33
7. Eiben AE, van Hemert JI, Marchiori E, Steenbeek AG (1998) Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In: Proceedings of the 5th international conference on parallel problem solving from nature (PPSN-V), LNCS 1498, pp 196–205
8. Garrett SM (2004) Parameter-free, adaptive clonal selection. In: Proceedings of the IEEE congress on evolutionary computation (CEC04), vol 1, pp 1052–1058
9. Mackworth AK (1977) Consistency in network of relations. Artificial Intelligence 8:99–118
10. Marchiori E (1977) Combining constraint processing and genetic algorithms for constraint satisfaction problems. In: Proceedings of the 7th international conference on genetic algorithms (ICGA97), pp 330–337
11. Nannen V, Eiben A (2007) Relevance estimation and value calibration of evolutionary algorithm parameters. Proceedings of the joint international conference for artificial intelligence (IJCAI), pp 975–980 (2007)
12. Riff MC (1998) A network-based adaptive evolutionary algorithm for csp. In: metaheuristics: advances and trends in local search paradigms for optimisation, chap 22. Kluwer Academic Publisher, Dordecht, pp 325–339
13. Riff MC, Zuniga M (2007) Towards an immune system to solve csp. Proceedings of the IEEE congress on evolutionary computation Singapur(In Press.)
14. Smith BM, Dyer ME (1996) Locating the phase transition in binary constraint satisfaction problems. Artificial Intelligence 81(1–2):155–181
15. Solnon C (2002) Ants can solve constraint satisfaction problems. IEEE Trans Evol Comput 6(4):347–357
16. Tsang EPK, Wang CJ, Davenport A, Voudouris C, Lau TL (1999) A family of stochastic methods for constraint satisfaction and optimization. In: Proceedings of the 1st international conference on the practical application of constraint technologies and logic programming (PACLP), London, pp 359–383