2

Evolutionary Computation in Constraint Satisfaction

Madalina Ionita, Mihaela Breaban and Cornelius Croitoru
Alexandru Ioan Cuza University, Iasi, Romania

1. Introduction

Many difficult computational problems from different application areas can be seen as constraint satisfaction problems (CSPs). Therefore, constraint satisfaction plays an important role in both theoretical and applied computer science.
Constraint satisfaction deals essentially with finding a best practical solution under a list of constraints and priorities. Many methods, ranging from complete and systematic algorithms to stochastic and incomplete ones, were designed to solve CSPs. The complete and systematic methods are guaranteed to solve the problems but usually perform a great amount of constraint checks, being effective only for simple problems. Most of these algorithms are derived from the traditional backtracking scheme. Incomplete and stochastic algorithms sometimes solve difficult problems much faster; however, they are not guaranteed to solve the problem even if given unbounded amount of time and space.
Because most of the real-world problems are over-constrained and do not have an exact solution, stochastic search is preferable to deterministic methods. In this light, techniques based on meta-heuristics have received considerable interest; among them, population-based algorithms inspired by the Darwinian evolution or by the collective behavior of decentralized, self-organized systems, were successfully used in the field of constraint satisfaction.
This chapter presents some of the most efficient evolutionary methods designed for solving constraint satisfaction problems and investigates the development of novel hybrid algorithms derived from Constraint Satisfaction specific techniques and Evolutionary Computation paradigms. These approaches make use of evolutionary computation methods for search assisted by an inference algorithm. Comparative studies highlight the differences between stochastic population-based methods and the systematic search performed by a Branch and Bound algorithm.

2. Constraint satisfaction

A *Constraint Satisfaction Problem (CSP)* is defined by a set of variables $X = \{X_1, \dots, X_n\}$, associated with a set of discrete-valued domains, $D = \{D_1, \dots, D_n\}$, and a set of constraints $C = \{C_1, \dots, C_m\}$. Each constraint C_i is a pair (S_i, R_i), where R_i is a relation $R_i \subseteq D_{S_i}$ defined on a subset of variables $S_i \subseteq X$ called the scope of C_i. The relation denotes all compatible tuples of D_{S_i} allowed by the constraint.

Source: New Achievements in Evolutionary Computation, Book edited by: Peter Korosec,
ISBN 978-953-307-053-7, pp. 318, February 2010, INTECH, Croatia, downloaded from SCIYO.COM

A solution is an assignment of values to variables $x = (x_1, \ldots , x_n)$, $x_i \in D_i$, such that each constraint is satisfied. If a solution is found, then the problem is named satisfiable or consistent. Finding a solution to CSP is a NP-complete task.
However, the problem may ask for one solution, all solutions, or - when a solution does not exist - a partial solution that optimizes some criteria is desired. Our discussion will focus on the last case, that is, the Max-CSP problem. The task consists in finding an assignment that satisfies a maximum number of constraints. For this problem the relation R_i is given as a cost function $C_i(X_{i1} = x_{i1}, \ldots ,X_{ik} = x_{ik}) = 0$ if $(x_{i1}, \ldots , x_{ik}) \in R_i$ and 1 otherwise. Using this formulation, an inconsistent CSP can be transformed into a consistent optimization problem.
There are two major approaches to solve constraint satisfaction problems: search algorithms and inference techniques (Dechter, 2003). Search algorithms usually seek for a solution in the space of partial instantiations. Because the hybrid methods presented in this chapter make use of inference techniques, we present next an introduction to directional consistency algorithms.

2.1 Inference: directional consistency

Inference is the process of creating equivalent problems through problem reformulation. The variable domains are shrunk or new constraints are deduced from existing ones making the problem easier to solve with search algorithms. Occasionally, inference methods can even deliver a solution or prove the inconsistency of the problem without the need for any further search.
Inference algorithms used to ensure local consistency perform a bounded amount of inference. The primary characteristic by which they are distinguished is the number of variables or the number of constraints involved. Any search algorithm will benefit from representations that have a high level of consistency. The complexity of enforcing *i*-consistency is exponential in *i*, as this is the time and space needed to infer a constraint based on *i* variables. There is a trade-off between the time spent on inference and the time spent on subsequent search.
Because of the nature of search algorithms which usually extend a partial solution in order to get a complete one, the notion of directional consistency was introduced. The inference is restricted relative to a given ordering of the variables. Directed arc-consistency is the simplest algorithm in this category; it ensures that any legal value in the domain of a single variable has a legal match in the domain of any other selected variable (Wallace, 1995; Larossa et al., 1999).

2.1.1 Bucket Elimination

Bucket Elimination (Dechter, 1999; 1996) is a less expensive directional consistency algorithm that enforces global consistency only relative to a certain variable ordering. The algorithm takes as input an ordering of variables and the cost functions. The method partitions the functions into buckets. Each function is placed in the bucket corresponding to the variable which appears latest in the ordering. After this step, two phases take place then. In the first phase the buckets are processed from last to first. The processing consists in a variable elimination procedure that computes a new function which is placed in a lower bucket. In the second phase, the algorithm considers the variables in increasing order. It builds a solution by assigning a value to each variable, consulting the functions created during the first phase.

Mini-bucket Elimination (MBE) (Dechter & Rish, 2003) is an approximation of the previous algorithm which tries to reduce space and time complexity. The buckets are partitioned into smaller subsets, called mini-buckets which are processed separately, in the same way as in BE. The number of variables from each mini-bucket is upper bounded by a parameter, *i*. The time and space complexity of the algorithm is $O(exp(i))$. The scheme allows this way adjustable levels of inference. This parameter controls the trade-off between the quality of the approximation and the computational complexity.
For the Max-CSP problem, the MBE algorithm produces new functions computed as the sum of all constraint matrices and minimizes it over the bucket's variable (Kask & Dechter, 2000).

Algorithm 1 Mini-Bucket Elimination(i)

```
Require: A constraint network $R$, an ordering $d = (x_1, \ldots, x_n)$
Ensure: The set of ordered augmented buckets, an upper bound on the Max-CSP value, and
  an assignment.

  Initialize: Partition constraints into buckets $bucket_1, \ldots bucket_n$:
  for $i \leftarrow n$ downto 1 do
    put in $bucket_i$ all constraints whose highest variable is $X_i$
  end for
  Let $S_1 \ldots S_j$ be the scopes be of functions (new or old) in the processed bucket

  Backward:
  for $p \leftarrow n$ downto 1 do
    for all functions $h_1, h_2, \ldots h_j$ in $bucket_p$ do
      if $bucket_p$ contains an instantiation $X_p = x_p$ then
        assign $X_p = x_p$ to each $h_i$ and put each resulting function into its appropriate bucket.
      else
        generate an ($i$)-partitioning $Q' = \{Q_1, \ldots, Q_r\}$
        for each $Q_l \in Q'$ containing $h_{l_1}, \ldots, h_{l_t}$ generate function $h^l = min_{X_p} \sum_{i=1}^{t} h_{l_i}$
        add $h^l$ to the bucket of the latest variable in
        $U_p \leftarrow \bigcup_{i=1}^{j} S(h_{l_i}) - \{X_p\}$
      end if
    end for
  end for

  Forward:
  for $i = 1$ to $n$ do
    given $x_1, \ldots, x_{p-1}$ choose a value $x_p$ of $X_p$ that minimizes the sum of all the cost functions
    in $X_p$'s bucket
  end for

  Output the ordered set of augmented buckets, an upper bound and a lower bound assign-
  ment.
```

The mini-bucket algorithm is expanded in (Kask & Dechter, 2000) with a mechanism to generate some heuristic functions. The functions recorded by MBE can be used as a lower bound for the number of constraints violated by the best extension of any partial assignment. For this reason these functions can be used as heuristic evaluations functions in search. Given a partial assignment of the first p variables $x^p = (x_1, \ldots, x_p)$, the number of constraints violated by the best extension of x^p is:

$$f^*(x^p) = min_{x_{p+1},\ldots,x_n} \sum_{k=1}^{n} C_k$$

for the variable ordering $d = (X_1, \ldots, X_n)$.
The previous sum can be computed as:

$$f^*(x^p) = \Big(\sum_{C_i \in buckets(1\ldots p)} C_i \Big)(x^p) + h^*(x^p)$$

where $h^*(x^p)$ can be estimated by a heuristic function $h(x^p)$, derived from the functions recorded by the MBE algorithm. $h(x^p)$ is defined as the sum of all the h_j^k functions that satisfy the following properties:

- they are generated in buckets $p + 1$ through n, and
- they reside in buckets 1 through p.

$$h(x^p) = \sum_{i=1}^{p} \sum_{h_j^k \in buckets_i h_j^k} h_j^k, \ where\ k > p$$

h_j^k represents the function created by processing the j-th mini-bucket in $bucket_k$. The heuristic function f can be updated recursively:

$$f(x^p) = f(x^{p-1}) + H(x_p),$$

where $H(x_p)$ computes the cost of extending the instantiation x^{p-1} with the value x_p for the variable placed on the position p in the given ordering:

$$H(x_p) = \sum_j C_{p_j}(x^p) + \Big(\sum_k h_{p_k}(x^p) - \sum_j h_j^p(x^p)\Big).$$

In the formula above C_{p_j} are the constraints in bucket p, h_{p_k} are the functions in bucket p and h_j^p are the functions created in bucket p which reside in buckets 1 through p -1.

3. Evolutionary algorithms for CSPs

3.1 Existing approaches

Evolutionary Computation techniques are population-based heuristics, inspired from the natural evolution paradigm. All techniques from this area operate in the same way: they maintain a population of individuals (particles, agents) which is updated by applying some operators according to the fitness information, in order to reach better solution areas. The most known evolutionary computation paradigms include evolutionary algorithms (Genetic Algorithms, Genetic Programming, Evolutionary Strategies, Evolutionary Programming) and swarm intelligence techniques (Ant Colony Optimization and Particle Swarm Optimization).

Evolutionary algorithms (Michalewicz, 1996) are powerful search heuristics which work with a population of chromosomes, potential solutions of the problem. The individuals evolve according to rules of selection and genetic operators.
Because the application of operators cannot guarantee the feasibility of offspring, constraint handling is not straightforward in an evolutionary algorithm. Several methods were proposed to handle constraints with Evolutionary algorithms. The methods could be grouped in the following categories (Michalewicz, 1995; Michalewicz & Schoenauer, 1996; Coello & Lechunga, 2002):

- preserving feasibility of solutions
  For particular problems, where the generic representation schemes are not appropriate, special representations and operators have been developed (for example, the GENOCOP (GEnetic algorithm for Numerical Optimization of COnstrained Problems) system (Michalewicz & Janikow, 1991). A special representation is used aiming at simplifying the shape of the search space. Operators are designed to preserve the feasibility of solutions.
  Other approach makes use of constraint consistency to prune the search space (Kowalczyk, 1997). Unfeasible solutions are eliminated at each stage of the algorithm. The standard genetic operators are adapted to this case.
  Random keys encoding is another method which maintains the feasibility of solutions and eliminates the need of special operators. It was used first for certain sequencing and optimization problems (Bean, 1994). The solutions encoded with random numbers are then used as sort keys to decode the solution.
  In the decoders approach (Dasgupta & Michalewicz, 2001), the chromosomes tell how to build a feasible solution. The transformation is desired to be computationally fast.
  Another idea, which was first named *strategic oscillation*, consists in searching the areas close to the boundary of feasible regions (Glover & Kochenberger, 1995).
- penalty functions
  The most common approach for constraint-handling is to use penalty functions to penalize infeasible solutions (Richardson et al., 1989). Usually, the penalty measures the distance from the feasible region, or the effort to repair the solution. Various types of penalty functions have been proposed. The most commonly used types are:
  - static penalties which remain constant during the entire process
  - dynamic functions which change through a run
  - annealing functions which use techniques based on Simulated Annealing
  - adaptive penalties which change according to feedback from the search
  - co-evolutionary penalties in which solutions evolve in one population and penalty factors in another population
  - death penalties which reject infeasible solutions.

  One of the major challenges is choosing the appropriate penalty value. Large penalties discourage the algorithm from exploring infeasible regions, and push rapidly the EA inside the feasible region. For low penalties, the algorithm will spend a lot of time exploring the infeasible region.
- repairing infeasible solution candidates
  Repair algorithms are problem dependent algorithms which modify a chromosome in such a way that it will not violate the constraints (Liepins & Vose, 1990). The repaired solution is used only for evaluation or can replace with some probability the original individual.

- separation of objectives and constraints
  The constraints and the objectives are handled separately. For example, in (Paredis, 1994) a co-evolutionary model consisting of two populations, one of constraints, one of possible solutions is proposed. The populations influences each other; an individual with a high fitness from the population of potential solutions represents a solution which satisfies many constraints; an individual with a high fitness from the population of constraints represent a constraint that is violated by many possible solutions.
  Another idea is to consider the problem as a multi-objective optimization problem, in which we will have m+1 objectives, m being the number of constraints. Then we can apply a technique from this area to solve the initial problem.
- hybrid methods
  Evolutionary algorithms are coupled with another techniques.

There have been numerous attempts to use Evolutionary algorithms for solving constraint satisfaction problems (Dozier et al., 1994), (Paredis, 1994), (Eiben & Ruttkay, 1996). The *Stepwise Adaptation of Weights* is one of the best evolutionary algorithms for CSP solving. The constraints that are not satisfied are penalized more. The weights are initialized (with 1) and reset by adding a value after a number of steps. Only the weights for the constraints that are violated by the best individual are adjusted. An individual is a permutation of variables. A partial instantiation is constructed by considering the variables for assigning values in the order given by the chromosome. The variable is left uninstantiated if all possible values add a violation. The uninstantiated variables are penalized. The fitness is equal with the sum of all penalties.
Another efficient approach is the *Microgenetic Iterative Descent Algorithm* (Dozier et al., 1994). The algorithm uses a small population size. At each iteration an offspring is created by crossover or mutation operator, the operator being chosen after an adaptive scheme. A candidate solution is represented by n alleles, a pivot and a fitness value. Each allele has the variable, its value, the number of constraint violations the variable is involved in and an *hvalue* used for initializing the pivot. The pivot is used to choose the variable that will undergo mutation. If the fitness of the child is worse than the parent value, the h-value of the pivot offspring is decremented. The pivot is updated next: for each allele, the sum of the number of constraint violations and its h-value are computed; the allele with the highest value is chosen as the pivot. The fitness function is adaptive, employing the Morris Breakout Creating Mechanism (Morris, 1993) to escape from local optima.
Another approach for solving CSPs makes use of heuristics inside the evolutionary algorithm. In (Eiben et al., 1994) heuristics are incorporated into the genetic operators. The mutation operator selects a number of variables to be mutated and assigns them new values. The selected variables are those appearing in constraints that are most often violated. The new values are those that maximize the number of satisfied constraints. Another way of incorporating heuristic information in an evolutionary algorithm is described in (Marchiori & Steenbeek, 2000). The heuristics are not incorporated into operators, but as a standalone module. Individual solutions are improved by calling a local optimization procedure for each of them and then blind genetic operators are applied.
In (Craenen et al., 2003) a comparison of the best evolutionary algorithms is given.

3.2 Hybrid evolutionary algorithms for CSP

Generally, to obtain good results for a problem we have to incorporate knowledge about the problem into the evolutionary algorithm. Evolutionary algorithms are flexible and can be

easily extended by incorporating standard procedures for the problem under investigation. The heuristic information introduced in an evolutionary algorithm can enhance the exploitation but will reduce the exploration. A good balance between exploitation and exploration is important.
We will describe next the approach presented in (Ionita et al., 2006). The method includes information obtained through constraint processing into the evolutionary algorithm in order to improve the search results. The basic idea is to use the functions returned by the minibucket algorithm as heuristic evaluation functions. The selected genetic algorithm is a simple one, with a classical scheme. The special particularity is that the algorithm uses the inferred information in a genetic operator and an adaptive mechanism for escaping from local minima.
A candidate solution is represented by a vector of size equal to the number of variables. The value at position *i* represents the value of the corresponding variable, x_i. The algorithm works with complete solutions, i.e. all variables are instantiated. Each individual in the population has associated a measure of its fitness in the environment. The fitness function counts the number of violated constraints by the candidate solution.
In an EA the search for better individuals is conducted by the crossover operator, while the diversity in the population is maintained by the mutation operator.
The recombination operator is a fitness-based scanning crossover. The scanning operator takes as input a number of chromosomes and returns one offspring. It chooses one of the *i*-th genes of the *n* parents to be the *i*-th gene of the offspring. For creating the new solution, the best genes are preserved. Our crossover makes use of the pre-processing information gathered with the inference process. It uses the functions returned by the mini-bucket algorithm, $f^*(x^p)$ to decide the values of the offspring. The variables are instantiated in a given order, the same as the one used in the mini-bucket algorithm. A new value to the next variable is assigned by choosing the best value from the parents according to the evaluation functions f^*. As stated before, these heuristic functions provide an upper bound on the cost of the best extension of a given partial assignment.

Algorithm 2 *multiparent_crossover*$(P(t), k)$

```
for each set of k parents, $p_1, \ldots, p_k$ do
    for each position i in the ordering do
        $child_i \leftarrow best(p_{1i}, \ldots, p_{ki})$
        /* use $f^*(p_1^i), \ldots, f^*(p_k^i)$ in best */
        $parent \leftarrow$ get_worst_parent$(p_1, \ldots, p_k)$
        replace$(parent, child)$
    end for
end for
```

This recombination operator intensifies the exploitation of the search space. It will generate new solutions if there is sufficient diversity in the population. An operator to preserve variation is necessary. The mutation operator has this function, i.e. it serves for exploration. The operator assigns a new random value for a given variable.
After the application of the operators, the new individuals will replace the parents. Selection will take place next to ensure the preservation of fittest individuals. A fitness-based selection was chosen for experiments.

Because the crossover and the selection direct the search to most fitted individuals, there is a chance of getting stuck in local minima. There is a need to leave the local minima and to explore different parts of the search space. Therefore, we have included the earliest breakout mechanism (Morris, 1993). When the algorithm is trapped in a local minimum point, a breakout is created for each nogood that appears in this current optimum. The weight for each newly created breakout is equal to one. If the breakout already exists, its weight is incremented by one. A predefined percent of the total weights (penalties) for an individual that violates these breakouts are added to the fitness function. In this manner the search is forced to put more emphasis on the constraints that are hard to satisfy. The evaluation function is an adaptive function because it is changed during the execution of the algorithm.

4. Particle swarm optimization for CSPs

The idea of combining inference with heuristics was also tested on another population-based paradigm, the Particle Swarm Optimization. The method presented in (Breaban et al., 2007) is detailed next.

Algorithm 3 GA-MBE(i)

```
apply MBE(i)
$t \leftarrow 0$
initialize $P(t)$
evaluate $P(t)$
while termination condition not meet do
  $t \leftarrow t+1$
  select $P(t)$ from $P(t-1)$
  multiparent_crossover ($P(t), k$)
  mutation($P(t)$)
  evaluate $P(t)$
  if no diversity then
    get the list of breakouts from the best individual
    introduce the breakouts in the fitness
  end if
end while
```

4.1 Particle swarm optimization

Particle Swarm Optimization is a Swarm Intelligence technique which shares many features with Evolutionary Algorithms. Swarm Intelligence is used to designate the artificial intelligence techniques based on the study of collective behavior in decentralized, self-organized systems. Swarm Intelligence systems are typically made up of a population of simple autonomous agents interacting locally with one another and with their environment. Although there is no centralized control, the local interactions between agents lead to the emergence of global behavior. Examples of systems like this can be found in nature, including ant colonies, bird flocking, animal herding, bacteria molding and fish schooling.
The PSO model was introduced in 1995 by J. Kennedy and R.C. Eberhart, being discovered through simulation of a simplified social model such as fish schooling or bird flocking (Kennedy & Eberhart, 1995). PSO consists of a group (swarm) of particles moving in the search space, their trajectory being determined by the fitness values found so far.

The formulas used to actualize the individuals and the procedures are inspired from and conceived for continuous spaces. Each particle is represented by a vector x of length n indicating the position in the n-dimensional search space and has a velocity vector v used to update the current position. The velocity vector is computed following the rules:

- every particle tends to keep its current direction (an inertia term);
- every particle is attracted to the best position p it has achieved so far (a memory term);
- every particle is attracted to the best particle g in population (the particle having the best fitness value); there are versions of the algorithm in which the best particle g is chosen from topological neighborhood.

Thus, the velocity vector is computed as a weighted sum of the three terms above. The formulas used to update each of the individuals in the population at iteration t are:

$$v_i^t = v_i^{t-1} + w_1 \cdot (p_i - x_i^{t-1}) + w_2 \cdot (g_i - x_i^{t-1}) \tag{1}$$

$$x_i^t = x_i^{t-1} + v_i^t \tag{2}$$

4.2 Adapting PSO to Max-CSP

Schoofs and Naudts (Schoofs & Naudts, 2002) have previously adapted the PSO algorithm for solving binary constraint problems. Our algorithm is formulated for the more general Max-CSP problem. The elements of the algorithm are presented below.
An individual is an instantiation of all variables with respect to their domains.
The evaluation (fitness) function counts the violated constraints. Because Max-CSP is formulated as a minimization problem smaller values of the evaluation function correspond to better individuals.
The algorithm uses the basic idea of PSO: every particle tends to move towards his personal best and towards the global best. Updating the particle consists in instantiating its variables by choosing from the values of the two particles or keeping its own values. The decision is made based on the values of the heuristic function described in section 2.1.1. The MBE inference scheme is used as a preprocessing step.
The velocity and the operators must be redefined in order to adapt the PSO formulas to the problem. This technique has already been used in order to adapt the PSO to discrete problems. For example, for permutation problems the velocity was redefined as a vector of transposition probabilities (X. Hu et al., 2003) or as a list of transpositions (Clerc, 2000) and the sum between a particle position and the velocity consists in applying the transpositions.
We define the velocity which results from the subtraction of two positions $\overrightarrow{v} = \overrightarrow{y} \ominus \overrightarrow{x}$ as the vector $\overrightarrow{v} = [x_i \rightarrow y_i]$ where $\rightarrow$ represents as in (Schoofs & Naudts, 2002) a change of position.
The sum $\overrightarrow{v_1} \circ \overrightarrow{v_2}$ of the two velocities $\overrightarrow{v_1} = \overrightarrow{p} \ominus \overrightarrow{x}$ and $\overrightarrow{v_2} = \overrightarrow{g} \ominus \overrightarrow{x}$ produces the velocity $\overrightarrow{u}$ given by

$$u_i = \begin{cases} p_i \ominus x_i, H(p_i) < H(g_i) \\ g_i \ominus x_i, otherwise \end{cases}$$

where H is the heuristic function described in section 2.1.1.
The addition $\overrightarrow{p} = \overrightarrow{x} \oplus \overrightarrow{v}$ of the velocity $\overrightarrow{v} = \overrightarrow{y} \ominus \overrightarrow{z}$ to a position $\overrightarrow{x}$ is defined by
$p_i = \begin{cases} y_i, x_i = z_i \\ x_i, otherwise \end{cases}$
No parameter is used.

The PSO formulas become:

$$\overrightarrow{v}^t = \overrightarrow{v}^{t-1} \circ (\overrightarrow{p} \ominus \overrightarrow{x}^{t-1}) \circ (\overrightarrow{g} \ominus \overrightarrow{x}^{t-1}) \quad (3)$$

$$\overrightarrow{x}^t = \overrightarrow{x}^{t-1} \oplus \overrightarrow{v}^t \quad (4)$$

Because the first term $\overrightarrow{v}^{t-1}$ in equation (3) is the velocity used to obtain the position $\overrightarrow{x}^{t-1}$ at time t -1 we replace it with the velocity $\overrightarrow{v}^{t-1} = \overrightarrow{x}^{t-1} \ominus \overrightarrow{x}^{t-1}$. In this way the resulted velocity formula selects the particle which has the smaller heuristic function value from the current position x, the personal best p and the global best g.
The pseudocode of the algorithm is illustrated in Algorithm 4.
The step (*) from the pseudocode can be described as:

```
find $minimum(H(x_i), H(p_i), H(g_i))$
if $minimum = H(p_i)$ then
  $x_i = p_i$
else if $minimum = f(g_i)$ then
      $x_i = g_i$
```

Algorithm 4 PSO-MBE

```
order the variables
apply MBE(i)
randomly initialize the population (swarm particles)
evaluate population
find the best individual $g$ in population and set personal best $p$
**while** stop criteria not met **do**
  **for all** particle $x$ in population **do**
    **for** $i = 1$ to $n$ **do**
      $x_i = x_i \oplus ((x_i \ominus x_i) \circ (p_i \ominus x_i) \circ (g_i \ominus x_i))$      $(*)$
    **end for**
    **if** $eval(x) \geq eval(g)$ **then**
      apply mutation to particle $x$
    **end if**
    update personal best $p$
  **end for**
  find the best individual $g$ in population
**end while**
```

In order to explore the search space and to prevent the algorithm from getting trapped in local optima a mutation operator is introduced. This is identical to the one used in GAs: a random value is set on a random position. The role of the mutation is not only to maintain diversity but also to introduce values from variables' domains which do not exist in the current population. To maintain diversity, the algorithm also uses the following strategies: 1. in case of equal values for the evaluation function the priority is given to the current value and then to the personal optimum; 2. the algorithm is not implementing the online elitism: the best individual is not kept in population, the current optimum can be replaced by a worst individual in future iterations.

5. Tests and results

5.1 Data suite

Experiments were conducted only on binary CSPs (each constraint is built over at most two variables), but there is no reason that the algorithm could not be run on *n*-ary CSPs with $n > 2$.

The algorithms were tested on two well-known models for generating CSPs.

The four-parameter model (Smith, 1994), called **model B** does not allow the repetition of the constraints. A random CSP is given by four parameters (N, K, C, T), where N represents the number of variables, K the domain size, C the number of constraints and T the constraint tightness. The tightness represents the number of tuples not allowed. C constraints are selected uniformly at random from the available $N(N - 1)/2$ ones and for each constraint T nogoods are selected from the available K^2 tuples. We have tested the approach on some over-constrained classes of binary CSPs. The selected classes are sparse $\langle 25, 10, 37, T\rangle$, with medium density $\langle 15, 10, 50, T\rangle$ and complete graphs $\langle 10, 10, 45, T\rangle$. For each class of problem the algorithms were tested on 50 instances.

We investigate the hybrid approaches also against the set of CSP instances made available by Craenen et al. on the Web[1]. These instances are generated using the **model E** (Achlioptas et al., 2001). We have experimented with 175 solvable problem instances: 25 instances for different values of p in model $E(20, 20, p, 2)$. Parameter p takes the following values: {0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.30}. All instances considered were solvable.

5.2 Algorithms settings

The variable ordering used in MBE was determined with the min-induced-width heuristic. This method places the variable with the minimum degree last in the ordering. It connects then all of the variable neighbors, removes the node and all its adjacent edges and next repeats the procedure.

Experiments were made for different levels of inference, changing the values of the parameter i in the MBE algorithm. Values 0 and 1 for parameter i means that no inference is used. Value 2 for i corresponds to a DAC (directed arc consistency) preprocessing adapted to Max-CSP: instead of removing values from variable domains cost functions are added for variable-value pairs that count the number of variables for which no legal value match is found. Greater values for parameter i generate new cost functions over at most i -1 variables.

For model B, for each class of problems 50 instances were generated. The problems were first solved using a complete algorithm PFC-MRDAC (Larossa & Meseguer, 1998). This algorithm is an improved branch-and-bound algorithm, specifically designed for the Max-CSP problem. The optimal solution was the solution found by the PFC-MRDAC algorithm.

For each instance, for both PSO-MBE and GA-MBE, five independent runs were performed.

The number of parents for the recombination operator in GA-MBE was established to five. The population size was set to 40 in the GA-MBE, while for PSO-MBE the swarm size was equal to 30 particles. A time limit of 30 seconds was imposed for all search algorithms (the time limit is used only for the search phase and does not include time needed for MBE).

For comparison purposes the Branch and Bound algorithm described in (Kask & Dechter, 2000) was implemented.

[1] 1http://www.xs4all.nl/ ̃bcraenen/resources/csps modelE v20 d20.tar.gz

5.3 Results

As measures of effectiveness we use as in (Craenen et al., 2003) the success rate and the mean error at termination. The success rate represents the percentage of runs that find a solution. The mean error at termination for a run is equal to the number of constraints which are violated by the best solution, at the end of the algorithm.
The average number of constraint checks and the average duration of the algorithms until the optimum solution is reached was recorded only for the runs which find the optimum within the time limit.

5.3.1 Results for MBE and Branch-and-Bound

The results concerning the two criteria on model B instances for the inference algorithm and a Branch and Bound algorithm are given in Table 1. Each line of the table corresponds to a class of CSPs.
Obviously, the Mini-bucket elimination algorithm solves more problems when the bound *i* increases. The Branch-and-Bound algorithm behaves similarly. However, the time needed to find a solution increases too (see Table 2).

Class	MBE B&B SR/ME $i=0$	MBE B&B SR/ME $i=2$	MBE B&B SR/ME $i=4$	MBE B&B SR/ME $i=6$
25-10-37-84	0/ 7.72 0.02/ 5.70	0/ 5.28 0.3/ 2.82	0.48/ 0.78 1/ 0	1/ 0 1/ 0
25-10-37-85	0/ 8.20 0.02/ 6.22	0/ 5.30 0.18/ 3.06	0.3/ 1.16 1/ 0	0.98/ 0.02 1/ 0
15-10-50-84	0/ 7.78 0/ 5.42	0/ 6.32 0.06/ 3.94	0/ 4.60 0.88/ 0.16	0.14/ 2.02 0.98/ 0.02
15-10-50-85	0/ 8.2 0/ 6.02	0/ 5.48 0.06/ 3.14	0.02/ 4.98 0.78/ 0.38	0.14/ 2.24 1/ 0
10-10-45-84	0/ 5.6 0.10/ 2.16	0/ 4.74 0.32/ 1.22	0/ 4.16 1/ 0	0.18/ 2.18 1/ 0
10-10-45-85	0.02/ 5.90 0.14/ 2.48	0.02/ 5.44 0.24/ 1.74	0.06/ 4.28 1/ 0	0.16/ 1.86 1/ 0

Table 1. Results on model B: the success rate and the mean error for Mini-Bucket Elimination (MBE) and Branch and Bound (B&B) algorithms for different levels of inference(i)

The results on model E are given in Figure 1 and Figure 2.
The Branch and Bound algorithm is not influenced at all by the low levels of inference performed (the three curves in Figure 1 and 2 overlap); higher levels of inference are necessary but they require much more time and space resources.

5.3.2 Results for the hybrid evolutionary computation algorithms Model B

The success rate and the mean error measures for the hybrid approaches are given in Table 3.
The search phase of the hybrid algorithms improves considerably the performance of the inference algorithm. For the class of problems 15 – 10 – 50 – 84 the MBE with *i* = 4 did not find the optimum for any of the generated problems. GA-MBE and PSO-MBE have solved 41%, respectively 52% of the problems.

Class	MBE B&B i=0	MBE B&B $i=2$	MBE B&B $i=4$	MBE B&B $i=6$
25-10-37-84	0.0 25.860	0.001 12.031	0.063 0.013	0.921 0.001
25-10-37-85	0.0 0.609	0.0 10.774	0.061 0.026	0.935 0.002
15-10-50-84	- -	0.0 10.906	0.115 6.250	11.924 1.357
15-10-50-85	- -	0.0 13.531	0.110 8.431	12.292 0.909
10-10-45-84	0.0 15.824	0.0 12.936	0.096 2.199	7.419 0.296
10-10-45-85	0.0 6.555	0.0 7.179	0.095 2.330	7.438 0.286

Table 2. Results on model B: average time in seconds for MBE and B&B algorithms for the runs which return the optimum

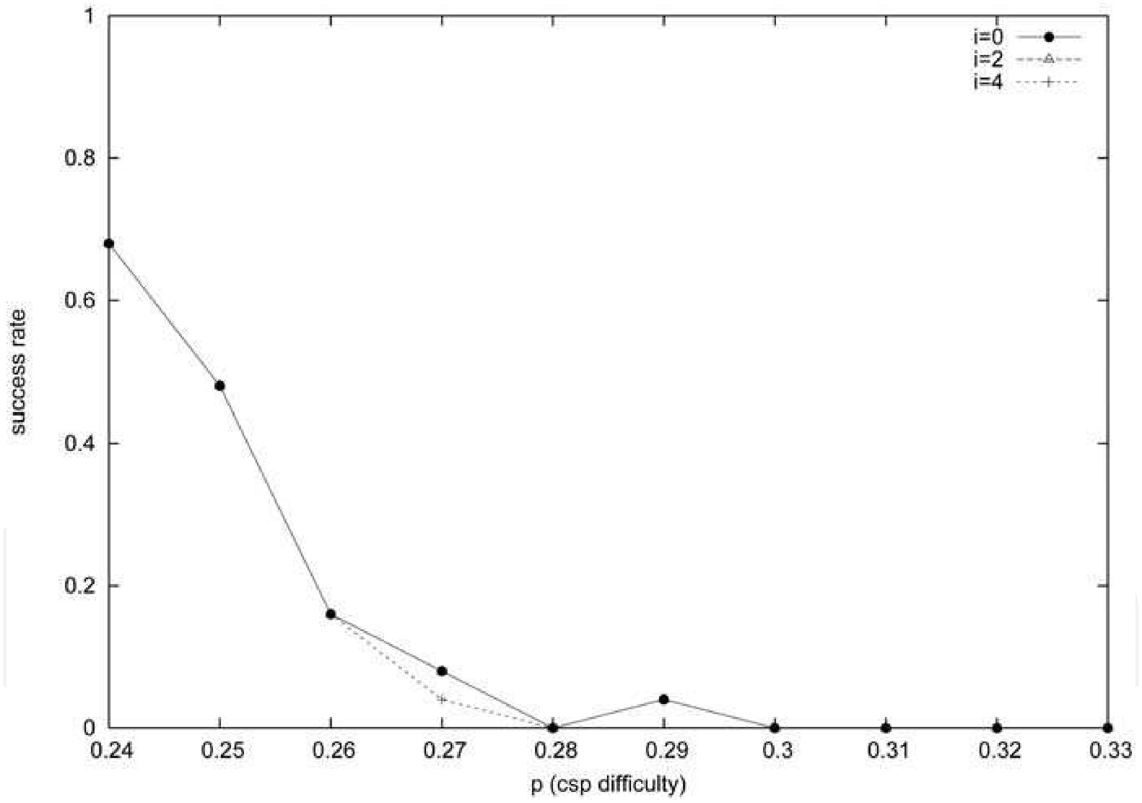

Fig. 1. Results on model E: Success rate for B&B

Even when the optimum solution was not found, the hybrid algorithms return a solution closed to the optimum. This conclusion can be drawn from Table 3 by looking at the mean error values.

We have also used an additional criterium for the evaluation of the hybrid algorithms. The standard measure of the efficiency of an evolutionary algorithm, the number of fitness evaluations is not very useful in this context. The use of heuristics implies more

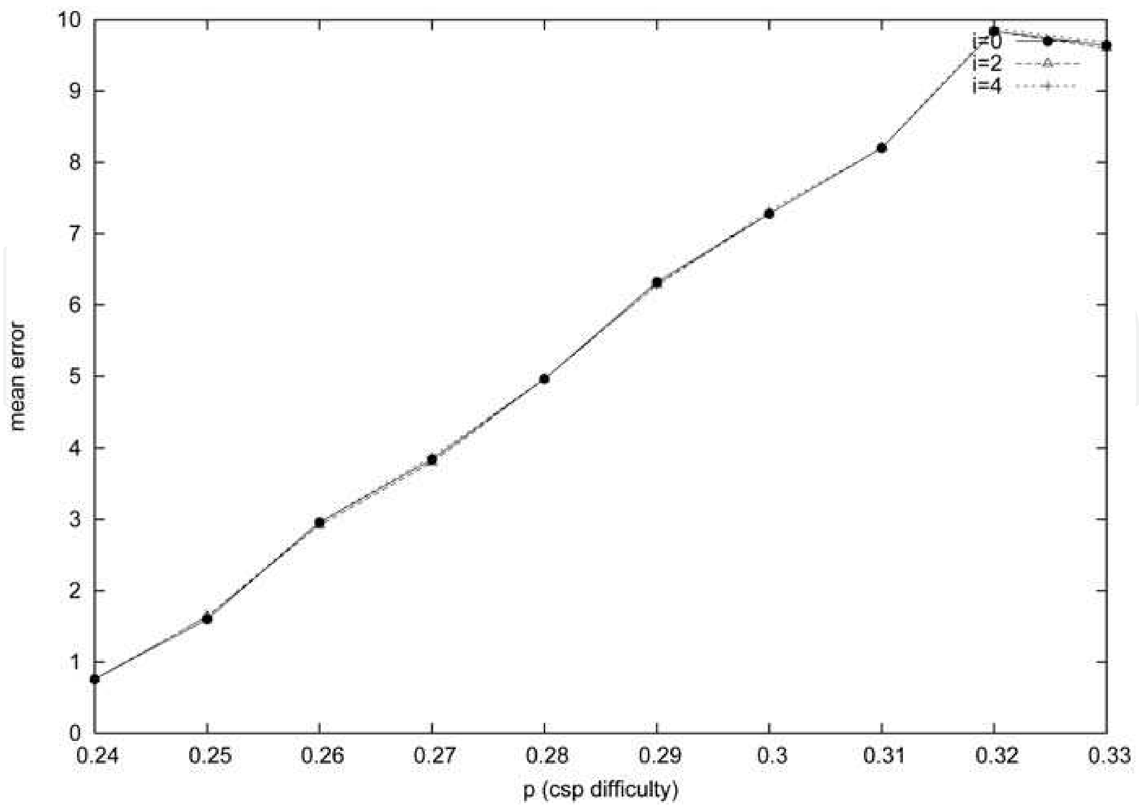

Fig. 2. Results on model E: Average mean error for B&B

Class	GA-MBE PSO-MBE SR/ME $i = 0$	GA-MBE PSO-MBE SR/ME $i = 2$	GA-MBE PSO-MBE SR/ME $i = 4$	GA-MBE PSO-MBE SR/ME $i = 6$
25-10-37-84	0.05/ 2.23 0.29/ 1.02	0.23/ 1.24 0.32/ 0.95	0.88/ 0.12 0.74/ 0.31	1/ 0 1/ 0
25-10-37-85	0.03/ 2.46 0.20/ 1.21	0.12/ 1.42 0.27/ 1.10	0.79/ 0.22 0.65/ 0.42	1/ 0 0.98/ 0.01
15-10-50-84	0.22/ 1.30 0.58/ 0.55	0.13/ 1.82 0.21/ 1.46	0.41/ 0.80 0.52/ 0.70	0.61/ 0.49 0.54/ 0.65
15-10-50-85	0.09/ 1.58 0.58/ 0.58	0.11/ 1.84 0.22/ 1.36	0.30/ 1.18 0.36/ 1.08	0.59/ 0.51 0.52/ 0.70
10-10-45-84	0.65/ 0.36 0.66/ 0.40	0.42/ 0.78 0.43/ 0.83	0.55/ 0.57 0.42/ 0.80	0.64/ 0.40 0.48/ 0.80
10-10-45-85	0.73/ 0.29 0.71/ 0.38	0.42/ 0.91 0.41/ 1.07	0.59/ 0.56 0.53/ 0.83	0.78/ 0.27 0.64/ 0.54

Table 3. Results on model B: the success rate and the mean error for GA and PSO hybrid algorithms

computation that is invisible for this metric. Therefore we have computed the average number of constraint checks, for the runs which return the optimum solution (see Table 4). Regarding the number of constraint checks performed by the two algorithms, one general rule can be drawn: the higher the inference level, the less the time spent on search.

Class	GA-MBE PSO-MBE $i = 0$	GA-MBE PSO-MBE $i = 2$	GA-MBE PSO-MBE $i = 4$	GA-MBE PSO-MBE $i = 6$
25-10-37-84	$5.5 \cdot 10^6$	$5.7 \cdot 10^6$	$5.6 \cdot 10^6$	$0.1 \cdot 10^6$
	$27.7 \cdot 10^6$	$10.0 \cdot 10^6$	$4.5 \cdot 10^6$	$1.5 \cdot 10^6$
25-10-37-85	$4.7 \cdot 10^6$	$8.6 \cdot 10^6$	$2.5 \cdot 10^6$	$0.1 \cdot 10^6$
	$22.0 \cdot 10^6$	$13.8 \cdot 10^6$	$2.6 \cdot 10^6$	$1.9 \cdot 10^6$
15-10-50-84	$22.4 \cdot 10^6$	$17.8 \cdot 10^6$	$5.3 \cdot 10^6$	$4.3 \cdot 10^6$
	$16.5 \cdot 10^6$	$23.0 \cdot 10^6$	$10.8 \cdot 10^6$	$6.4 \cdot 10^6$
15-10-50-85	$24.5 \cdot 10^6$	$14.0 \cdot 10^6$	$11.6 \cdot 10^6$	$4.3 \cdot 10^6$
	$25.4 \cdot 10^6$	$10.9 \cdot 10^6$	$11.4 \cdot 10^6$	$3.5 \cdot 10^6$
10-10-45-84	$18.5 \cdot 10^6$	$11.8 \cdot 10^6$	$14.5 \cdot 10^6$	$7.6 \cdot 10^6$
	$16.6 \cdot 10^6$	$10.8 \cdot 10^6$	$5.26 \cdot 10^6$	$3.5 \cdot 10^6$
10-10-45-85	$10.8 \cdot 10^6$	$16.9 \cdot 10^6$	$11.5 \cdot 10^6$	$6.28 \cdot 10^6$
	$9.37 \cdot 10^6$	$9.84 \cdot 10^6$	$4.85 \cdot 10^6$	$1.07 \cdot 10^6$

Table 4. Results on model B: the average constraint checks of the hybrid evolutionary computation algorithms

For sparse instances the efficiency of the preprocessing step is evident for the two algorithms: increasing the inference level more problems are solved. The Genetic Algorithm for medium density cases behaves similarly as for the sparse one. For complete graphs, the genetic algorithm for i = 0 (no inference) gives a good percent of solved problems. But the best results are for the larger level of inference. In almost all cases, the performance of the GAMBE is higher when using a higher i-bound. This proves that the evolutionary algorithm uses efficiently the information gained by preprocessing.

When combined with PSO, inference is useful only on sparse graphs; on medium density and complete graphs low levels of inference slow down the search process performed by PSO and the success rate is smaller. Higher levels of inference (i = 6) necessitate more time spent on preprocessing and for complete graphs it is preferable to perform only search and no inference.

Unlike evolutionary computation paradigms, the systematic Branch and Bound algorithm has much benefit from inference preprocessing for all classes of problems. When no inference is performed B&B solves only 2% of the sparse instances and 14% of the complete graphs. The approximative solutions returned after 30 seconds run are of lower quality than those returned by the evolutionary computation methods (the mean error is high). When inference is used the turnaround becomes obvious starting with value 4 for parameter i.

Table 5 lists the average time spent by MBE and PSO algorithms for the runs which return the optimum. Similarly, Table 6 refers to MBE and B&B time. These tables are illustrative for the inference/search trade-off: increasing the inference level the time needed by the search algorithms to find the optimum decreases.

An interesting observation can be drawn regarding the time needed by PSO to find the optimum: even if the algorithm is run for 30 seconds the solved instances required much shorter time; this is a clear indicator that PSO is able to find good solutions in a very short time but it gets stuck often in local optima and further search is compromised.

Class	MBE PSO i=0	MBE PSO $i=2$	MBE PSO $i=4$	MBE PSO $i=6$
25-10-37-84	0 8.746	0.003 3.917	0.099 0.824	1.081 0.558
25-10-37-85	0 8.910	0.004 6.087	0.096 0.654	1.115 0.877
15-10-50-84	0 3.467	0.003 5.549	2.384 2.384	12.906 1.123
15-10-50-85	0 4.488	0 3.490	0.176 2.490	13.780 0.963
10-10-45-84	0 2.529	0.019 1.806	0.111 0.986	7.511 0.701
10-10-45-85	0 1.529	0.001 1.703	0.111 0.939	7.774 0.222

Table 5. Results on model B: average time in seconds for MBE and PSO algorithms for the runs which return the optimum

Model E

The results for model E corresponding to GA-MBE are given in Figure 3 and 4. Figures 5 and 6 present the results for PSO-MBE.

Class	MBE BB i=0	MBE BB $i=2$	MBE BB $i=4$	MBE BB $i=6$
25-10-37-84	0.0 25.860	0.001 12.031	0.063 0.013	0.921 0.001
25-10-37-85	0.0 0.609	0.0 10.774	0.061 0.026	0.935 0.002
15-10-50-84	- -	0.0 10.906	0.115 6.250	11.924 1.357
15-10-50-85	- -	0.0 13.531	0.110 8.431	12.292 0.909
10-10-45-84	0.0 15.824	0.0 12.936	0.096 2.199	7.419 0.296
10-10-45-85	0.0 6.555	0.0 7.179	0.095 2.330	7.438 0.286

Table 6. Results on model B: average time in seconds for MBE and B&B algorithms for the runs which return the optimum

The performance of the algorithms decreases with the difficulty of the problem. For smaller values of p (0.24) the percentage of solved problems increases with the inference level. For more difficult problems low levels of inference are useless.
One can also observe that the mean error is small, meaning that the algorithm is stable (Figure 4 and Figure 6). This feature is very important for such kind of problems.
Given that the bounded inference performed on the model E instances has low effect on subsequent search both for the randomized and the systematic methods, GA-MBE and

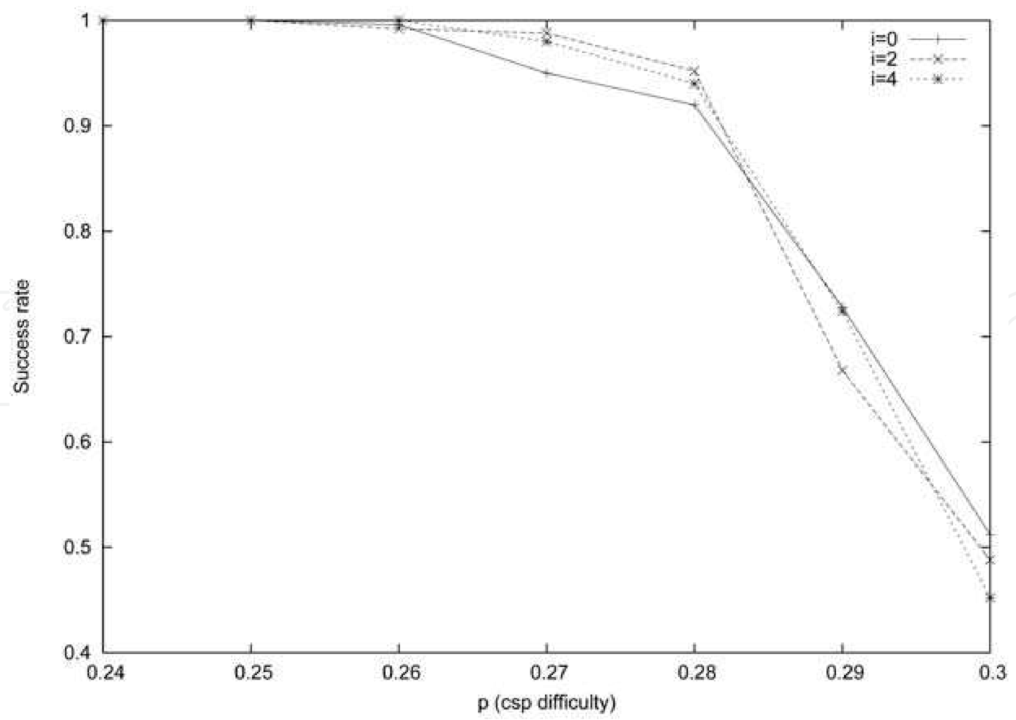

Fig. 3. Results on model E: success rate for GA-MBE

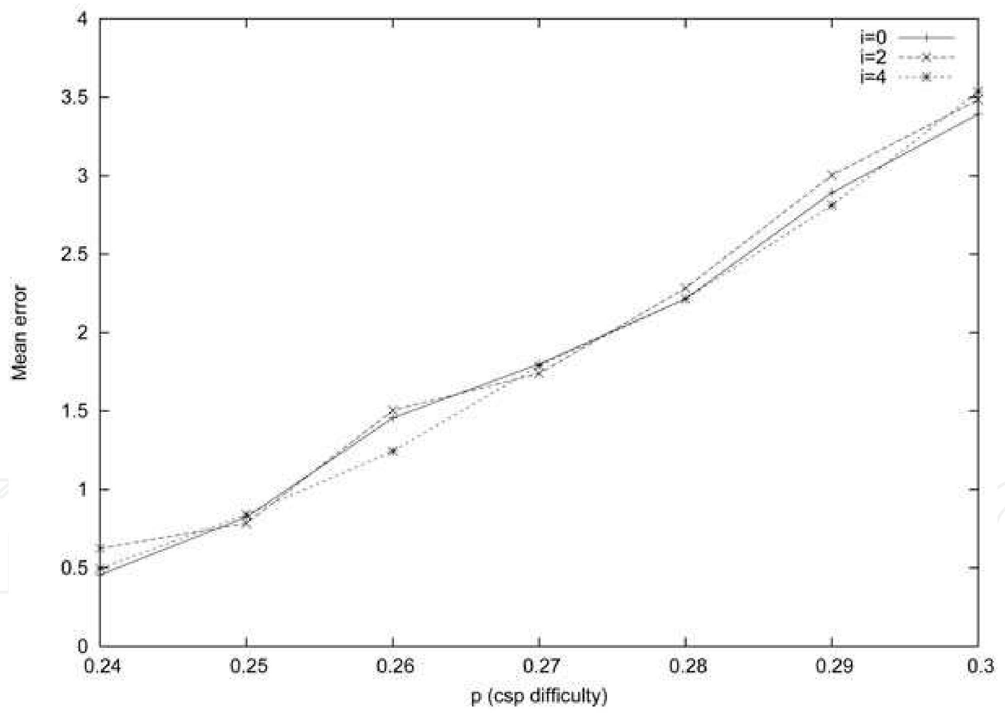

Fig. 4. Results on model E: mean error for GA-MBE

PSOMBE obtain better results than B&B: the percentage of solved problems (SR) is higher and the approximative solutions returned after 30 seconds run are better qualitatively. The average number of constraint checks on model E test instances increases with parameter p from $5 \cdot 10^7$ to $8 \cdot 10^7$ for PSO-MBE and from $9 \cdot 10^7$ to $2 \cdot 10^8$ for B&B. The average time for the runs which find the optimum increases with p from 9 seconds to 13 seconds for PSO-MBE and from 10 seconds to 18 seconds for B&B.

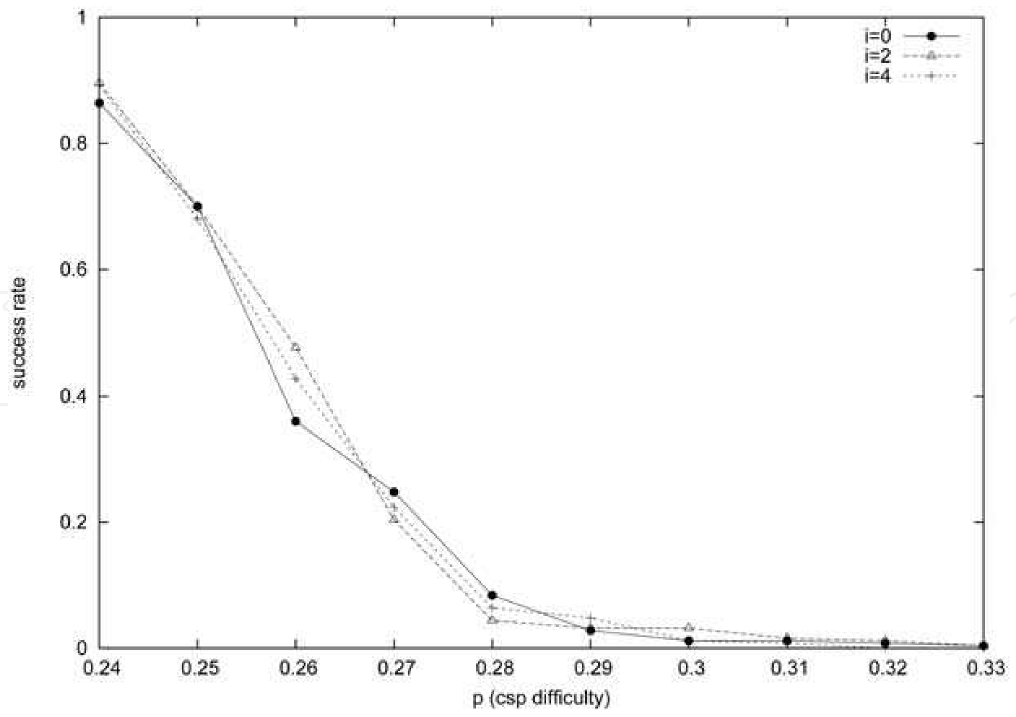

Fig. 5. Results on model E: success rate for PSO-MBE

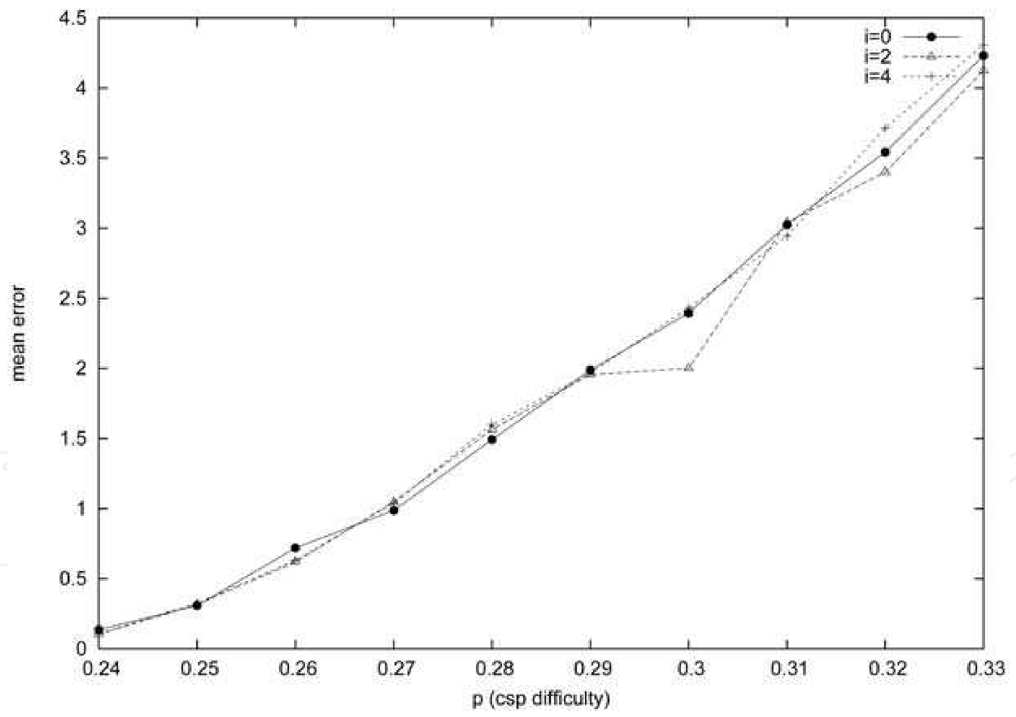

Fig. 6. Results on model E: mean error for PSO-MBE

Using the results from the comparative study of several genetic algorithms made by Craenen et al. (Craenen et al., 2003) we can conclude that the performance of the hybrid algorithms is comparable with that of the best GAs in the CSP field: *Stepwise Adaptation of Weights* and *Glass-Box GA*. Low levels of inference slightly improve the performance of our algorithm on difficult CSP instances; higher levels of inference are needed.

6. Conclusion

The chapter presents some of the techniques based on Evolutionary Computation paradigms for solving constraints satisfaction problems. Two hybrid approaches based on the idea of using the heuristics extracted from an inference algorithm inside evolutionary computation paradigms are detailed. The effect of combining inference with randomized search was studied by exploiting the advantage of adaptable inference levels offered by the Mini-Bucket Elimination algorithm. Tests conducted on binary CSPs against a Branch and Bound algorithm show that the systematic search has more benefit from inference than the randomized search performed by evolutionary computation paradigms. However, on hard CSP instances the Branch and Bound algorithm requires higher levels of inference which imply a much greater computational cost in order to compete with evolutionary computation methods.

7. References

Achlioptas, D., Kirousis, L., Kranakis, E., Krizanc, D., Molloy, M. & Stamatiou, Y. (2001). Random constraint satisfaction: A more accurate picture, *Constraints* 4(6): 329–344.

Bean, J. (1994). Genetic algorithms and random keys for sequencing and optimization, *ORSA Journal on Computing* 6(2): 154–160.

Breaban, M., Ionita, M. & Croitoru, C. (2007). A new pso approach to constraint satisfaction, *Proceedings of IEEE Congress on Evolutionary Computation*, pp. 1948–1954.

Clerc, M. (2000). Discrete particle swarm optimization. illustrated by the traveling salesman problem, *Technical report*. Available at htpp://www.mauriceclerc.net.

Coello, C. & Lechunga, M. (2002). Mopso: A proposal for multiple objective particle swarm optimization, *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 1051–1056.

Craenen, B., Eiben, A. & van Hemert, J. (2003). Comparing evolutionary algorithms on binary constraint satisfaction problems, *IEEE Transactions on Evolutionary Computation* 7(5): 424–444.

Dasgupta, D. & Michalewicz, Z. (2001). *Evolutionary Algorithms in Engineering Applications*, Springer 1st edition.

Dechter, R. (1996). Bucket elimination: A unifying framework for probabilistic inference algorithms, *Uncertainty in Artificial Intelligence*, pp. 211–219.

Dechter, R. (1999). Bucket elimination: A unifying framework for reasoning, *Artificial Intelligence* 113(1-2): 41–85.

Dechter, R. (2003). *Constraint Processing,* Morgan Kaufmann Publishers.

Dechter, R. & Rish, I. (2003). Mini-buckets: A general scheme for bounded inference, *Journal of the ACM* 50(2): 107–153.

Dozier, G., Bowen, J. & Bahler, D. (1994). Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithm, *Proceedings of the 1st IEEE Conference on Evolutionary Computation*, pp. 306–311.

Eiben, A., Raue, P.-E. & Ruttkay, Z. (1994). Solving constraint satisfaction problems using genetic algorithms, *Proceedings of the 1st IEEE Conference on Evolutionary Computation*, pp. 542–547.

Eiben, A. & Ruttkay, Z. (1996). Self-adaptivity for constraint satisfaction: Learning penalty functions, *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, pp. 258– 261.

Glover, F. & Kochenberger, G. (1995). Critical event tabu search for multidimensional knapsack problems, *Proceedings of the International Conference on Metaheuristics for Optimization*, pp. 113–133.

Ionita, M., Croitoru, C. & Breaban, M. (2006). Incorporating inference into evolutionary algorithms for max-csp, *Lecture Notes in Computer Science. 3rd International Workshop on Hybrid Metaheuristics* 4030: 139–149.

Kask, K. & Dechter, R. (2000). New search heuristics for max-csp, *Lecture Notes In Computer Science. Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming* 1894: 262–277.

Kennedy, J. & Eberhart, R. (1995). Particle swarm optimization, *Proceedings of the 1995 IEEE International Conference on Neural Networks*, Vol. 4, pp. 1942–1948.

Kowalczyk, R. (1997). Constraint consistent genetic algorithms, *Proceedings of the IEEE International Conference on Evolutionary Computation*, pp. 343–348.

Larossa, J. & Meseguer, P. (1998). Partial lazy forward checking for max-csp, *Proceedings of the 13th European Conference on Artificial Intelligence*, pp. 229–233.

Larossa, J., Meseguer, P. & Schiex, T. (1999). Maintaining reversible dac for max-csp, *Artificial Intelligence* 107(1): 149–163.

Liepins, G. & Vose, M. (1990). Representational issues in genetic optimization, *Journal of Experimental and Theoretical Artificial Intelligence* 2(2): 101–115.

Marchiori, E. & Steenbeek, A. (2000). A genetic local search algorithm for random binary constraint satisfaction problems, *Proceedings of the 14th Annual Symposium on Applied Computing*, pp. 458–462.

Michalewicz, Z. (1995). A survey of constraint handling techniques in evolutionary computation methods, *Proceedings of the 4th Anual Conference on Evolutionary Programming*, pp. 135–155.

Michalewicz, Z. (1996). *Genetic Algorithms + Data structures = Evolution programs*, Springer Berlin 3rd edition.

Michalewicz, Z. & Janikow, C. Z. (1991). Handling constraints in genetic algorithms, *Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 151–157.

Michalewicz, Z. & Schoenauer, M. (1996). Evolutionary algorithms for constrained parameter optimization problems, 4(1): 1–32.

Morris, P. (1993). The breakout method for escaping from local minima, *Proceedings of the 11th National Conference on Artificial Intelligence*, pp. 40–45.

Paredis, J. (1994). Coevolutionary constraint satisfaction, *Lecture Notes In Computer Science. Proceedings of the 3rd Conference on Parallel Problem Solving from Nature* 866: 46–55.

Richardson, J., Palmer, M., Liepins, G. & Hilliard, M. (1989). Some guidelines for genetic algorithms with penalty functions, *Proceedings of the 3rd International Conference on Genetic Algorithms*, p. 191197.

Schoofs, L. & Naudts, B. (2002). Swarm intelligence on the binary constraint satisfaction problem, *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 1444–1449.

Smith, B. (1994). Phase transition and the mushy region in constraint satisfaction, *Proceedings of the 11th ECAI*, pp. 100–104.

Wallace, R. (1995). Directed arc consistency preprocessing, *Lecture Notes in Computer Science. Constraint Processing, Selected Papers* 923: 121–137.

X. Hu, X., Eberhart, R. & Shi, Y. (2003). Swarm intelligence for permutation optimization: a case study on n-queens problem, *Proceedings of the IEEE Swarm Intelligence Symposium*, pp. 243–246.