Improving Evolutionary Algorithms for Efficient Constraint Satisfaction

Peter Stuckey and Vincent Tam

Department of Computer Science
The University of Melbourne
Parkville 3052, Australia

Abstract

Hard or large-scale constraint satisfaction and optimization problems, occur widely in artificial intelligence and operations research. These problems are often difficult to solve with global search methods, but many of them can be efficiently solved by local search methods. Evolutionary algorithms are local search methods which have considerable success in tackling difficult, or ill-defined optimization problems. In contrast they have not been so successful in tackling constraint satisfaction problems. Other local search methods, in particular GENET and EGENET are designed specifically for constraint satisfaction problems, and have demonstrated remarkable success in solving hard examples of these problems. In this paper we examine how we can transfer the mechanisms that were so successful in (E)GENET to evolutionary algorithms, in order to tackle constraint satisfaction algorithms efficiently. An empirical comparison of our evolutionary algorithm improved by mechanisms from EGENET and shows how it can markedly improve on the efficiency of EGENET in solving certain hard instances of constraint satisfaction problems.

1 Introduction

A *constraint satisfaction problem* (CSP) (see e.g. [19, 11]) can be specified as a triple $\langle Z, D, C\rangle$ involving a finite set $Z = \{X_1, \ldots, X_n\}$ of variables, a function D which maps each variable $X_i \in Z$ to a finite set (domain) D_i of possible values, and a finite set $C = \{c_1, \ldots, c_m\}$ of constraints defined on these variables. CSPs naturally occur in many important industrial applications, such as planning, scheduling and resource allocation. A *binary CSP* is a CSP where each constraint involves at most two variables. Given a CSP, we try to find a solution, if one exists, by assigning values, from their associated domains, to all the variables so that no constraint in C is violated.

CSPs are, in general, NP-complete. Thus a general algorithm designed to solve this class of problems may require exponential time in the worst case. In practice, there are two main approaches to solve CSPs. Global search methods involve some form of backtracking search that systematically explores the entire search space. Therefore they ensure that a solution will be found if one exists. However they can be slow on solving certain large-scale or hard instances of CSPs [9, 14, 19].

On the other hand, local search methods, such as artificial neural networks, evolutionary algorithms and simulated annealing, begin from a valuation and make local improvements by reassigning values to variables until a good solution is found. Local search methods sacrifice completeness, that is they may not be able to find a solution to a CSP even when one exists. However, local search methods based on ideas such as the min-conflict heuristic have recently been shown to be more efficient than the global search methods on solving some large-scale or hard instances of real-life CSPs [7, 14, 1].

The min-conflict heuristic [14] forms the basis for many global and local search methods. The idea behind the min-conflict heuristic is to consider modifying only a single variable at a time, and to assign a value to that variable which is locally minimum in terms of constraint violations. When there are several local minima (ties) in terms of constraint violations, a value will randomly be selected among these ties. Thus, the min-conflict heuristic performs the steepest-descent step after considering every possible value in the domain of the selected variable. The min-conflict step is repeatedly applied for all variables in turn. The min-conflict heuristic has been used in a number of successful local search methods, for example [14] shows that it can solve the million-queens problem in minutes.

GENET [3] is a min-conflict based artificial neural network for solving binary CSPs. It has had remarkable success in solving certain hard CSPs such as hard graph coloring problems. Lee *et. al.* [9] extended GENET to EGENET which incorporates a generic constraint representation scheme for handling general CSPs. EGENET has also been successfully applied to solve general CSPs such as car-sequencing problems and cryptarithmetic problems in an efficient manner. Both GENET and EGENET combine a min-conflict based state update rule with a heuristic learning rule. Initially, a complete and random variable assignment is generated. Then, the network executes a convergence procedure as follows. Each variable is asynchronously updated by the min-conflict heuristic in each convergence cycle. When there is no change in any value assigned to the variables, the network is trapped in a local minima. If the local minimum does not represent a solution, a heuristic learning rule, is applied to help the network escape from these local minima. The network convergence procedure iterates until a solution is found or a predetermined resource limit is exceeded.

Evolutionary algorithms are examples of local search methods designed principally for tackling optimization problems. They consider a population of candidate solutions, and by ranking these with a fitness function (the function to optimize), generate a population of offspring from the fitter individuals. As generations proceed the population moves toward better solutions. A constraint satisfaction problem can be considered as an optimization problem, by defining the fitness function in terms of the number of constraints violated by a valuation. However evolutionary algorithms typically perform poorly on constraint satisfaction problems unless they are specialised for this class.

Naively applying evolutionary algorithm approaches to CSPs using the above formulation is not efficient. When evolutionary algorithms are applied to solve CSPs with a large number of variables (say > 500), the total computational cost for checking violation of constraints for each candidate solution x in the population can be very large. This computational cost can be minimized by focusing the search only on a reasonably small population of chromosomes without much impact on the search efficiency. This is the idea behind micro-genetic algorithms, the subclass of evolutionary algorithms based on a small population size (usually < 20). Micro-genetic algorithms can find solutions with fewer iterations than evolutionary algorithms with larger population sizes for some problems [10, 6].

To further improve the search efficiency of micro-genetic algorithms in solving CSPs, researchers have tried adding different heuristics in the evolutionary computation. For instance, Rojas [13] used a heuristic to define the importance of a constraint in a constraint network on which the fitness function is based. With this more detailed fitness function she was able to improve an evolutionary algorithm in solving a set of randomly generated 3-colouring graphs. On the other hand, Dozier *et al.* [6] proposed an interesting heuristic inheritance mechanism for their micro-genetic algorithm for tackling binary CSPs. The mechanism tries to minimize the number of constraint violations by continuously mutating only a single selected variable, or moving to mutate another variable. Further, they extended this micro-genetic algorithm with the integration of the Iterative Descent Method [12], which modifies the fitness function to consider not only the number of constraint violations but to also penalise revisiting inconsistent pairs of values (nogoods) found at previous local minima during the search.

Even integrated with these useful heuristics, most micro-genetic algorithms, like the conventional evolutionary algorithms, still depend heavily on the selection criteria of the evaluation function to guide the parallel local searches towards the global optimum of the fitness function, which in this case represents a solution to the CSP. On solving some moderately or highly constrained real-life CSPs, this approach can be slowed down due to the small-sized population of candidate solutions in the micro-genetic algorithm.[1]

In this paper we consider how to improve a micro-genetic algorithm in order to solve CSPs efficiently. Given that (E)GENET has been so successful in solving CSPs it seems attractive to integrate the min-conflict heuristic into the evolutionary algorithm in order to improve its search performance.

Because the min-conflict heuristic descends so quickly, it can easily be trapped in local minima. (E)GENET incorporates a heuristic learning rule to escape from local mimima. Our micro-genetic algorithm incorporating the min-conflict heuristic is also liable to be trapped in local minima. Thus we propose two dif-

[1] Because of the small population size, the selection pressure exerted by the fitness function is small. In the other words, the mating pool becomes less competitive with a small population size. Therefore, the evolutionary search tends to be *localized* in some sense.

ferent heuristic operators either to avoid or escape from these local minima. The first, population-based learning, is a generalization of the (E)GENET learning rule. The second, look-forward, in effect explores a broader local neighbourhood in order to avoid falling in local minima. To evaluate their performance, we built prototypes for these different proposals and compared them with EGENET on a number of CSPs. Our preliminary experimental results show that these heuristics are particularly useful in solving some hard instances of CSPs.

The paper is organised as follows. In the next section we briefly introduce evolutionary algorithms and the local search algorithm EGENET. Section 3 explores different uses of heuristics in micro-genetic algorithms to solve CSPs efficiently. We examine different possible mutation operators, the most successful genetic operator in solving CSPs, and then define the pivot gene concept of Dozier *et al.* Then, we define our generic min-conflict based micro-genetic algorithm, and explain two heuristic improvements operators that we can add to this algorithm: look-forward and population-based learning. In Section 4 we discuss and analyse experimental results on a number of different hard CSPs, comparing Dozier *et al*s micro-genetic algorithm and EGENET, both implemented as specific instances of our generic algorithm, versus the generic algorithm improved with different combinations of our proposed heuristic operators. Finally, we give some concluding remarks and suggest some possible future work in Section 5.

2 Preliminaries

2.1 Evolutionary algorithms

```
EA($PZ, MZ, fitness()$)
    initialize a *Population* of $PZ$ chromosomes
    **repeat**
        select the best $MZ$ chromosomes $\in$ *Population* according to $fitness()$
        $Population := \emptyset$
        **repeat**
            apply genetic operators to produce $\{offspring\}$
            $Population := Population \cup \{offspring\}$
        **until** $(sizeof(Population) = PZ)$
    **until** (*Population* is converged or resource limit is exceeded)
```

Figure 1: The convergence procedure of a generic evolutionary algortihm

Figure 1 shows the pseudo-code of a basic evolutionary algorithm. Given the population size PZ, the size MZ of the mating pool and the evaluation function $fitness()$, the evolutionary algorithm sets up an initial population accordingly. In each generation, the best MZ chromosomes, according to $fitness()$, are selected from the current *Population* to construct the mating pool in which some genetic operators such as mutation or crossover are applied to produce offspring to form the next generation. This "selection-and-reproduction" process is repeated until all the chromosomes have converged to the same local minima, or some predetermined resource limit is exceeded. A resource limit is usually defined in terms of CPU time or the maximum number of generations allowed. The result of the algorithm is the best solution found at the end of the computation.

Typical genetic operators are: mutation, where in a single chromosome a randomly selected gene is altered in some way; or crossover, where two genes are selected as parents that generate children by combining genetic information from the parents.

In order to tackle a CSP with an evolutionary algorithm, it is encoded in the following way. The value of each variable X_i in the CSP is represented by a *gene* in the evolutionary algorithm which is an element of D_i. Each *chromosome* is a sequence of genes corresponding to a valuation for all the variables. If $Z = \{X_1, \ldots, X_n\}$, then a chromosome $d_1 d_2 \cdots d_n$, $d_i \in D_i$ represents the valuation $(X_1 = d_1, \ldots, X_n = d_n)$

A solution to a CSP is a valuation which violates no constraint, so we can translate a CSP into the problem of minimizing the number of constraint violations. The fitness function required for this problem is defined by modelling constraints c_i as (satisfiability) functions U_i where

$$U_i(x) = \begin{cases} 0 & \text{if } x \text{ is a solution of } c_i \\ 1 & \text{otherwise} \end{cases}$$

and defining the fitness function as

$$\mathit{violations}(x) = \sum_{i=1}^{m} U_i(x)$$

where m is the number of constraints. The minimization problem is then

$$\min_{x \in D^n} \mathit{violations}(x)$$

$D^n = D_1 \times \ldots \times D_n$ is the Cartesian product of the (finite) domains for all the n variables.

Clearly for this form of problem, if we ever find a valuation z such that $\mathit{violations}(z) = 0$ we can terminate the evolutionary algorithm since we know it is a solution representing a global minimum.

2.2 EGENET

Lee and Won [8] proposed EGENET, an extension of GENET to handle arbitrary (non-binary) constraints. EGENET is, like GENET, a min conflict search with heuristic learning, but it differs from GENET in the representation of constraints and the learning rules. In fact it is now increasingly clear (see [4]) that both GENET and EGENET are discrete Lagrangian multiplier methods [17]. In EGENET there are two types of nodes: *variable nodes* and *constraint nodes*. Each variable X_i in a CSP is represented by a variable node with its associated domain. The *state* of a variable node X_i is defined to be its current value assignment $(X_i = v_i)$.

Each constraint node denotes a constraint c in the CSP. Let $vars(c)$ be the set of variables occurring in constraint c. A constraint node c is connected to a variable node X_i if the variable $X_i \in vars(c)$.

A constraint c, where $vars(c) = \{X_{i_1}, \ldots, X_{i_k}\}$, is considered as a mapping from valuations of the form $(X_{i_1} = v_{i_1}, \ldots, X_{i_k} = v_{i_k})$ to non-negative numbers (penalty values)[2]. If the constraint c holds for $(X_{i_1} = v_{i_1}, \ldots, X_{i_k} = v_{i_k})$ then $c(X_{i_1} = v_{i_1}, \ldots, X_{i_k} = v_{i_k}) = 0$. Otherwise, if the constraint c does not hold for the valuation $(X_{i_1} = v_{i_1}, \ldots, X_{i_k} = v_{i_k})$, then $c(X_{i_1} = v_{i_1}, \ldots, X_{i_k} = v_{i_k}) > 0$. We can naturally extend the notion of constraints as mapping to any valuation x which gives values to all the variables $\{X_{i_1}, \ldots, X_{i_k}\}$ and possibly others. For the remainder of the paper we shall only consider valuations x which give values to all the variables in the CSP. The constraint c maps valuations to a penalty value, which is 0 if the constraint is satisfied by the valuation, and otherwise some positive number. Note the difference from the satisfiability functions U_i above which only return 0 if satisfied or 1 if unsatisfied.

EGENET also acts as a constraint violation minimization approach, but the function that is minimized is more complicated. The penalty function penalises each constraint by some measure of the degree it is violated, and uses a changing penalty multiplier. For example, the penalty value for an inequality $e > 0$ (where e is some arithmetic expression) under valuation θ may be defined as: 0 if e's value under θ, $\theta(e)$ is greater than 0, and $p(1 - \theta(e))$ otherwise, where p is a (positive) penalty multiplier. The penalty value for a conjunction of constraints is defined as the sum of the penalties for each constraint. The definition of the penalty function is thus

$$penalty(x) = \Sigma_{i=1}^{m} p_i c_i(x)$$

where p_i is the penalty multiplier for the i^{th} constraint.

Figure 2 shows the constraint network and the EGENET network of a binary CSP with three variables X, Y and Z each with domain $\{1, 2, 3, 4\}$, and a set of constraints $\{X > 1, X < Y, X \neq Z, Y < Z\}$. The variables X, Y and Z are represented by variable nodes. Each variable nodes is initially set to a random element of the domain of the variable. The current valuation $(X = 1, Y = 2, Z = 3)$ can be read from these nodes.

The first constraint node $X > 1$ is connected to the variable node X. The binary constraint $X < Y$ is represented by the second constraint node, and is connected to variable nodes X and Y. Each constraint node shows the penalty value of the corresponding constraint for the current variable assignment. For

[2]We use positive penalties instead of the negative penalties used in the original EGENET model.

instance, the penalty value of the unary constraint $X > 1$ for the current variable assignment of $(X = 1)$ is $+1$. Assuming the penalty multipliers for the constraints are $\{2, 1, 3, 1\}$ respectively, the

$$penalty(X = 1, Y = 2, Z = 3) = 2 \times 1 + 1 \times 0 + 3 \times 0 + 1 \times 0 = 2$$

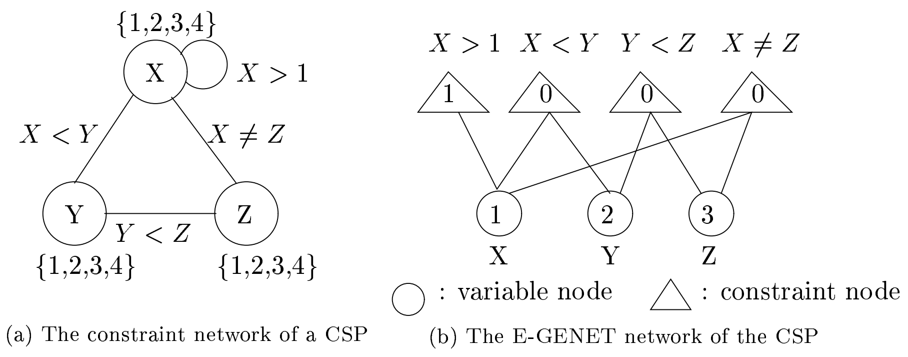
{1,2,3,4}
X
$X > 1$
$X < Y$
$X \neq Z$
Y
Z
$Y < Z$
{1,2,3,4}
{1,2,3,4}
(a) The constraint network of a CSP
$X > 1$ $X < Y$ $Y < Z$ $X \neq Z$
1
0
0
0
1
2
3
X
Y
Z
: variable node
: constraint node
(b) The E-GENET network of the CSP

Figure 2: The constraint network and EGENET network of an example CSP.

Each value v in the domain D_i of the variable X_i has an *input* value defined as the sum of penalty values of all the constraint nodes involving variable X_i when $X_i = v$. A positive input indicates the value v is involved in the violation of certain constraints. When the sum of inputs for the current variable assignments is zero, the network state represents a solution.

EGENET executes a network convergence procedure as follows. Given a current valuation x of the form $(X_1 = v_1, \ldots, X_n = v_n)$ and some variable X_i the input value is calculated for each valuation obtained by setting $X_i = v$ for each $v \in D_i$. The value v with minimum input is the new state for X_i. In the case of ties, if the old state v_i has minimum input then it remains the state of the node, otherwise one of values with minimum input is chosen randomly.

For example, updating X in the state illustrated in Figure 2, given the penalty multipliers define above $\{2, 1, 3, 1\}$, evaluates the penalty function for each of the possible values for X. When $X = 1$, the input (penalty) is 2, when $X = 2$ the penalty is 1, when $X = 3$ the penalty is 3, and when $X = 4$ the penalty is 3. Hence $X = 2$ is the minimum penalty value which will be the new value chosen for X.

The convergence procedure of EGENET is shown in Figure 3. The heuristic learning rule in EGENET is left unspecified intentionally since application of domain knowledge can help to design good learning rules for specific problems. For example, one heuristic rule is to increase the penalty multiplier of a violated constraint by 1 each time the learning rule is invoked.

```
repeat
    update all variable nodes in parallel asynchronously until no repair occurs
    if (sum of inputs for the network state is zero)
        terminate with success
    else
        activate heuristic learning
until (resource limit is exceeded)
```

Figure 3: The convergence procedure of EGENET

3 A generic micro-genetic algorithm for solving CSPs

This section begins by exploring the kinds of mutation operators which are likely to help in solving CSPs, and then examining the idea of pivot gene. We then define our generic micro-genetic algorithm making use of these concepts. To improve the algorithm further we propose a look-forward algorithm which is an intelligent search strategy to "explore" more opportunistic improvements, and a learning mechanism to help escape local minima and avoid re-exploring search space that has already been explored.

3.1 Genetic operators for solving CSPs

The most important form of genetic operator for tackling CSPs in evolutionary algorithms appears to be mutation operators. Most crossover operators are not very appropriate for generic CSP problems because they rely on a locality of information which is not usually present in the chromosomal representation of a valuation for a generic CSP. Hence single-point or double-point crossovers are not likely to be successful. Rojas [13] noted this problem, and added a permutation operation to her genetic approach to solving CSPs, specifically to modify the order in the chromosomal representation of a valuation. Because the complex interdependence of gene values in typical CSPs multi-point crossover also tend to destroy too much information to be useful in tackling CSPs.[3]

In contrast, a number of mutation operators are worth considering. The simplest is simply a single point mutation (single_pt_mutate). A child chromosome is obtained by replacing a selected gene in the parent by a random value from the domain of the corresponding variable. Given the difficulty in solving CSPs, it is usually disadvantageous to lose hard won information during a search, hence single_pt_mutate can be weak because the child chromosome may be significantly worse than its parent. Hence another possibility is descent_single_pt_mutate where single_pt_mutate is first applied, but if the fitness

[3]We made a number of experiments using crossover operators but they were uniformly unsuccessful in improving our micro-genetic algorithm.

of the parent is better than the child, then the child is replaced by a clone of the parent. The descent_single_pt_mutate is a form of probabilistic descent operation, which has been used successfully in other local search methods such as IDM [12]. Pseudo-code for single_pt_mutate and descent_single_pt_mutate is given in Figure 4. The arguments are x the parent chromosome to be mutated, i the number of the gene selected for mutation, and D the domain for that gene. The functions return the child chromosome. The function *random* selects a random element from a set. We assume a chromosome is represented by an array of genes. Note that the descent_single_pt_mutate function intentionally assigns a new value to a gene even when the fitness value of the new value is the same as that of the previously assigned value (as opposed to min_conflict_mutate below) in order to explore other parts of the search space.

```
single_pt_mutate($x, i, D$)
    $x[i] := random(D)$
    return $x$

descent_single_pt_mutate($x, i, D$)
    $z :=$ single_pt_mutate($x, i, D$)
    if $(fitness(z) \leq fitness(x))$ then
        return $z$
    else return $x$
```

Figure 4: Pseudo-code for the single point mutations.

The min-conflict heuristic can be considered as a single point mutation operator since it changes the value of a single variable (gene). The difference is that it considers all possible values for that gene and selects one of the most fit. Figure 5 gives the pseudo-code for the min-conflict mutation function, min_conflict_mutate. The function min_conflict_mutate is basically the same as the variable updating function applied to every variable in an EGENET convergence cycle. Here, we will only apply it to a single gene in each iteration. Given a chromosome x and gene i it generates all the chromosomes x' that result from modifying x so that the i^{th} gene takes every possible value v in its domain D. The set *ties* is the subset of D with best fitness (least violations). When the old value of the i^{th} gene is not bettered by any other value, then it is the result of the mutation is no change (thus the operation is stable). Otherwise a random element of *ties* is used in the resulting chromosome. Stability has been shown to be advantageous [3] when using the min-conflict heuristic.

3.2 Pivot gene selection

We could apply the above mutation operations to a randomly selected gene, or a set of randomly selected genes. But because of the difficult nature of CSPs it

```
min_conflict_mutate($x, i, D$)
    $old\_value := x[i]$
    $min\_fit := fitness(x)$
    $ties := \{old\_value\}$
    foreach $v \in D - \{old\_value\}$
        $x[i] := v$
        $curr\_fit := fitness(x)$
        if $(curr\_fit < min\_fit)$ then
            $min\_fit := curr\_fit$
            $ties := \{v\}$
        else
            if $(ties \neq \{old\_value\} \wedge curr\_fit = min\_fit)$ then
                $ties := ties \cup \{v\}$
            endif
        endif
    endforeach
    $x[i] := random(ties)$
    return $x$
```

Figure 5: Pseudo-code for the min-conflict mutation function.

is worth concentrating on the genes that are causing constraint violations. We can count the number of constraint violations in a chromosome x involving the variable X as

$$viol(X, x) = \Sigma\{U_i(x) \mid 1 \leq i \leq m, X \in vars(c_i)\}$$

Examining the genes that are involved in the most constraint violations is a simple approach, but clearly we may also have to consider other genes, even those not involved in any current constraint violations to increase the likelihood of finding a solution (otherwise we may not explore enough of the search space). In this section we consider mechanisms for selecting the gene to mutate, the *pivot* gene.

Dozier *et al.* [6] proposed a heuristic inheritance mechanism for determining the pivot gene in their approach to solving CSPs through a micro-genetic algorithm. Each gene (variable) in a chromosome has an attached hereditary value (h-value). The heuristic inheritance mechanism uses a pivoting scheme as follows. The pivoting scheme always choose the gene with the largest sum of constraint violations and h-value as the pivot gene. Initially, all the h-values are set to 0. In subsequent generations, the hereditary value is updated as follows: if the child created from parent chromosome has a better fitness value than its parent, then it inherits the same h-value as its parents. Otherwise the h-value of the child is one less than that of the parent. This approach means that while a pivot gene is successful in minimization then it will continue being used. But when it no longer improves fitness, eventually the negative h-value will lead to other genes being used as pivots. After some number of generations (called the heuristic reset rate) all the h-values of all the genes are reset to 0. This is required otherwise the h-values dominate the constraint violations in the selection process.

We consider two simpler pivot selection mechanisms. As in the proposal of Dozier *et al* the selection is based on the number of constraint violations and a hereditary value (h-value) for each gene. The first selection function usage-select is based on the usage of each gene as a pivot gene in the previous generations. When a gene is used as a pivot gene, its h-value will be set to 1. In each generation, usage-select only consider the unused genes (with h-value $= 0$) and chooses the one with the largest number of constraint violations. When there is a tie between the unused genes, the ties are broken randomly. Eventually every gene has been used as a pivot gene and all h-values are reset to 0. The second selection mechanism update_select is based on the effects of reassigning values to the genes. It only considers the genes (variables) possibly affected by updates of other genes in previous generations. The function update_select selects the gene with h-value equal to 0 and the maximum number of constraint violations. As in the previous mechanism, ties are broken randomly. Whenever a gene is updated, its h-value is set to 1, and each other gene which occurs in a constraint with the updated gene has its h-value reset to 0. Hence it will be reconsidered as a pivot gene.

3.3 A micro-genetic algorithm based on min-conflict

We can now define a generic micro-genetic algorithm for tackling CSPs using the above mutation and selection operators. The function MCHMGA takes as input a CSP $\langle Z, D, C \rangle$, a fitness function, $fitness$, together with a population size PZ and maximum number MAX_GENS of generations allowed for finding a solution to the CSP. The function returns a triple of the best fitness value $best\_fit$, the resulting population P with PZ chromosomes and the number $ngens$ of generations used. Whenever MCHMGA returns the $best\_fit$ equal to zero, there are no constraint violations and hence, there is, at least, one solution existing in the population P. Otherwise if it returns a positive value for $best\_fit$, the evolutionary algorithm has failed to find a solution.

MCHMGA is an instance of EA where only mutation operators are used. Initially, all the chromosomes in the population P of the evolutionary algorithm MCHMGA are assigned values randomly, and their initial h-values are set to zero by initialize_h-values. Then, each chromosome x is treated in turn. It is evaluated according to the fitness function $fitness$. If the fitness is zero, then a solution has been found and the algorithm returns the current population. Otherwise, the function select_pivot selects an important variable (the pivot gene) for the chromosome, to which to apply the min-conflict based mutation min_conflict_mutate, as well as updating the related h-values. All other variables are first possibly updated using the mutation function descent_single_pt_mutate. The procedure iterates until a solution is found, or the number $ngens$ of gener-

```
MCHMGA( $\langle Z, D, C\rangle, fitness, PZ, MAX\_GENS$ )
    Initialize a population $P$ of $PZ$ chromosomes by random variable assignments;
    $ngens := 0$
    $best\_fit := fitness(random(P))$
    foreach chromosome $cs \in P$ /* set all the h-values to 0 */
        initialize_h-values($cs$)
    endforeach
    while ($ngens < MAX\_GENS$)
        $P' := \emptyset$
        foreach chromosome $cs \in P$
            if ($fitness(cs) < best\_fit$) then
                $best\_fit := fitness(cs)$
            endif
            if ($best\_fit = 0$) then
                return $\langle 0, P, ngens\rangle$ /* solution found */
            endif
            $i :=$ select_pivot($cs$)
            foreach $1 \leq j \leq n$ where $j \neq i$
                $cs :=$ single_pt_mutate($cs, j, D_j$)
            endforeach
            $cs :=$ min_conflict_mutate($cs, i, D_i$)
            $P' := P' \cup \{cs\}$
        endforeach
        $P := P'$
        $ngens := ngens + 1$
    endwhile
    return $\langle best\_fit, P, MAX\_GENS\rangle$; /* returns sub-optimal solution */
```

Figure 6: Pseudo-code for the generic min-conflict based micro-genetic algorithm

ations used exceeds the predetermined limit MAX_GENS.

Unlike most evolutionary algorithms MCHMGA does not ever rank the population or apply crossover operations to create children from two or more parents. Instead each chromosome is modified to create its successor. The descending nature of our basic mutation operators, min_conflict_mutate and descent_single_pt_mutate, automatically ensures that a (single) offspring will always have no worse fitness value than its parent. In this way, our evolutionary algorithm MCHMGA can be considered as using a special selection mechanism which allows only offspring with no worse fitness than their parents to enter into the population. Otherwise, the selection function just replicates the parent into the population. When the best fitness value of the population in MCHMGA remains the same for consecutive evaluations, it is more likely that the "fittest" chromosome(s) in the population represents some local minima.

3.4 Look-forward search

Clearly since the min-conflict search is descending the MCHMGA algorithm can be trapped in local minima. Hence, we consider examining a wider set of neighbouring points in order to avoid the local minima. We call this *looking a step forward* because it examines the possible improvements we could obtain if we chose another value for the pivot gene and then mutated other variables further. When another value could lead to a globally better chromosome if other genes were also modified then this chromosome is the result of look-forward. This, in effect, wider local search can escape local minima that would trap the min-conflict based search because it looks further away from the current chromosome.

```
look_forward($x, i, D$)
    $candidates := ties - \{x[i]\}$
    if ($candidates = \emptyset$) then
        $candidates :=$ set of values $v \in D$ which give
                         second best fitness value when $x[i] = v$
    endif
    $best\_chrom := x$
    $best\_fit := fitness(x)$
    foreach $u \in candidates$
        $x[i] :=$ u
        foreach $1 \leq j \leq n$ where $j \neq i$
            $x :=$ single_pt_mutate($x, j, D_j$);
        endforeach
        $new\_fit := fitness(x)$;
        if ($new\_fit \leq best\_fit$) then
            $best\_fit := new\_fit$;
            $best\_chrom := x$;
        endif
    endforeach
    $x := best\_chrom$;
    return $x$
```

Figure 7: Pseudo-code for the look-forward algorithm.

Figure 7 shows the pseudo-code for the look-forward algorithm look_forward which applies to each chromosome in the population individually. The look_forward algorithm is inserted in MCHMGA directly after the call to min_conflict_mutate It examines the set *ties* built in min_conflict_mutate. If the set is a singleton, that is there is only one value for the pivot gene with best fitness, it sets *candidates* to all the values for the i^{th} gene with second best fitness. Otherwise *candidates* is set to *ties* with the current value of the i^{th} gene removed.

For each element in *candidates* a small local search is performed to see if it can lead to a better solution. Starting from a candidate, for each gene other than the pivot gene a descent_single_pt_mutate operation is applied. If a new chromosome is generated with an equal or better fitness than the result after min_conflict_mutate then it is the value returned. Looking forward in this manner is feasible because the mutation operator single_pt_mutate is computationally cheap. Also the set *candidates* tends to be small. It can destroy the stability property of min_conflict_mutate, but this may be required to escape a local minima.

We also use a a simplified version of the look-forward algorithm (denoted lazy_look_forward) which is only invoked when the best fitness value of the pivot gene is already zero (violating no constraint) and there are alternative values collected in the array *ties* for further consideration. This means there is no broader search of the neighbourhood except for genes and values which already satisfy all constraints. It is at this point where sideways moves can be most valuable.

3.5 Heuristic learning

```
popu_learn($P$)
    **foreach** $1 \leq i \leq m$
        $violated := false$;
        **foreach** $cs \in P$
            **if** $(c_i(cs) > 0)$ **then**
                $violated := true$;
            **endif**
        **endforeach**
        **if** $(violated = true)$ **then**
            $p_i := p_i + 1$
        **endif**
    **endforeach**
    **return**
```

Figure 8: Pseudo-code for the population-based learning.

We generalize the EGENET learning mechanism as *population-based learning* to improve the evolutionary search. Figure 8 shows the pseudo-code for the population-based learning method (popu_learn). It assumes that the fitness function being used is the *penalty* function defined in the previous section.

The basic idea is that when the search becomes trapped at a local minimum the violated constraints at this minimum are penalized more. This changes the landscape of fitness values and pushes search away from the local minimum. In MCHMGA learning is based on the whole population instead of the individual candidate solution. Initially, the penalty values for all constraints are set to 1. Whenever the whole population cannot be further improved by the local improvement operators, that is all are staying at some local minima (equilibrium point), the function popu_learn is invoked to check each constraint c against the current variable assignments in the chromosomes of the whole population. If any of these variable assignments violates the constraint c then the penalty value for c is increased by 1. The popu_learn function is inserted in MCHMGA at the end of each generation, immediately before the statement $P := P'$ as follows:

if $(P = P')$ **then** popu_learn

When the population size PZ equals 1 in MCHMGA, the function popu_learn is the same as the heuristic learning method used in GENET and EGENET (assuming a simple incrementing of the penalty multipliers). Hence, EGENET can be regarded as a specific instance of our generic MCHMGA with $PZ = 1$ and improved with popu_learn, where single_pt_mutate is not used and pivot genes are selected by usage-select. Note that population based learning provides the only communication between chromosomes, and is what makes our approach more than simply a parallel search.

4 Experimental Results

To demonstrate the efficiency of our approach, we built a prototype implementation with GCC version 2.7.2 on Digital Unix version 3.2F. We compare EGENET implemented as a variation of MCHMGA (so that variation in coding is minimized) against different versions of MCHMGA on a number of CSPs. We allows generations times population size to reach at most 1,000,000 for each algorithm, after which we terminate the execution with failure. EGENET executes with a population size $PZ = 1$ (and hence 1,000,000 generations), while for the other algorithms $PZ = 6$ (the same population size as that used in [6]). Benchmarks were executed on a DEC AlphaServer 8400 running at 300Mhz. The reported CPU time is measured in seconds. All the data are averages over 10 runs.

We consider two different versions of EGENET, with a fixed order of selecting pivot genes (EGENET-FIX) and with a random order of selecting pivot genes (EGENET-RND). The versions of MCHMGA we consider are those improved with the just the population-based learning method (PL), or population-based learning method and look_forward (PL+LF) or lazy_look_forward (PL+LLF). The benchmarks we consider are N-queens, a standard benchmark from the literature and a set of hard graph-colouring problems and a hard job-shop scheduling problems. All the benchmarks are executed using the update_select pivoting strategy, because it uniformly improved upon usage_select. We also tried incorporating a number of versions of multiple-point crossover operator into our generic MGA. But they all failed to solve any CSPs in our benchmarks.

N	EGENET-FIX/RND	PL	PL + LF	PL + LLF
10	0.110s/0.050s	0.120s	0.364s	0.064s
20	1.020s/0.960s	1.790s	3.090s	1.150s
30	1.740s/0.990s	14.3s	4.500s	1.010s
40	1.380s/0.910s	35.6s	6.560s	0.930s
50	1.210s/1.230s	79.2s	4.490s	1.280s
60	0.910s/1.050s	219.6s	4.540s	1.040s
70	0.800s/0.970s	378.4s	4.570s	1.300s
80	1.240s/1.860s	768.8s	3.800s	1.020s
90	1.210s/1.530s	921.1s	7.490s	0.920s

Table 1: Results for the N-Queens problems

We also implemented the two different micro-genetic algorithms defined in [6]. Unfortunately we could not recreate the results reported in [6]. Although we have carefully constructed our versions of their algorithms using all the information given in the paper, it appears that some significant difference must have remained. Our implementations of these algorithms were unable to solve any of the hard example of CSPs we consider, and were substantially slower on all the N-Queens examples.

4.1 N-Queens Problems

The N-Queens problem is the problem of placing N queens on an $N \times N$ chessboard so that no queen can take another. A queen attacks another queen when both of them are placed on the same row, column or diagonal. The N-Queens problem is a standard benchmark used by many researchers to determine the effectiveness of a search algorithm.

Table 1 shows the timings of all the different algorithms on the N-Queens problems where N varies from 10 to 90. The EGENET solvers and the versions of MCHMGA except (PL) have comparably good performance, showing the efficiency of min-conflict based variable updating together with heuristic learning in solving these CSPs. Clearly population based learning alone is not efficient in solving these problems. MCHMGA using population-based learning and look-forward is, on the average, about 3 to 7 times slower that those of the EGENET solvers, presumably because the look-forward algorithm directs the search to some unpromising branches of the search tree. In contrast, the lazy-look-forward algorithm improved the search of MCHMGA in solving these N-Queens problems. The PL + LLF solver always obtained the timings close to, and in some cases better than, those of the EGENET solvers.

4.2 Hard Graph Colouring Problems

The following hard graph colouring problems on which we compared the different algorithms are obtained from [5]. These problems have been shown to defeat many algorithmic complete search methods.

Nodes	Colours	EGENET-FIX/RND	PL	PL + LF	PL + LLF
125	18	7628s/9209s	3397s	7572s	1638s
125	17	47.63hr/137.9hr	failed	failed	37.39hr

Table 2: Results on a set of hard graph colouring problems

Table 2 shows the average timings required for each of our tested MGAs to solve these problems. The version of MCHMGA improved with popu_learn and lazy_look_forward performs the best among algorithms since the lazy look-forward algorithm is invoked less frequently than the full look-forward algorithm. EGENET-FIX betters EGENET-RND on these problems with some particular fixed ordering of variables for updating. PL and PL+LF both failed to find a solution (before exhausting the generation limit) in the harder case. This indicates again that population based learning alone may not be enough to compete with EGENET, and that their is a disadvantage of the frequent invocations of the full look-forward algorithm in MCHMGA. It appears the wider local search leads to some exploration of some unpromising parts of the search space for certain hard instances of CSPs, so that it is fact overall a disadvantage to use it.

4.3 Hard Job-shop Scheduling Problems

The benchmark problems are taken from a set of job-shop scheduling problems [15] from Carnegie Mellon University. These problems have been widely used for comparison of the performance of different schedulers. The goal for each problem is to fulfil all operations with a given time bound.

Table 3 shows the timings required for the different algorithms to solve these hard problems. For each case, the problem size ($m \times n$) is shown, where m denotes the number of jobs in each problem and n stands for the number of tasks contained in each job. For all the problems, EGENET-RND spends about $\frac{1}{4}$ of the time required by EGENET-FIX showing the importance of pivot gene selection on the performance of EGENET computation.

For these problems we considered an additional operation, bounds propagation (see e.g. [11]). Bounds propagation removes values from the domains of variables which are not bounds consistent. It is applied before the various algorithms are applied to the problem. The reduced domains of variables reduces the search space. The earlier benchmarks are bounds consistent (and indeed arc consistent) and hence the preprocessing step would not alter them in any way.

Table 4 shows the time required by the different algorithms when bounds propagation is used to reduce the problems. The bounds propagation technique improved the performance of all the algorithms, since for these problems it can prune the search space by a significant amount. When applied to the EGENET solvers, the bounds propagation technique achieved a more significant improvement compared to the MCHMGA algorithm. Presumably this is because EGENET is more sensitive to the domain sizes of the variable since it only ever uses the min_conflict_mutate operation which examines every possible

Problem	EGENET-FIX/RND	PL	PL + LF	PL + LLF
sched_la03 (10x5)	114.3s/13.29s	1.980s	1.850s	1.340s
sched_la04 (10x5)	241.5s/10.90s	1.810s	1.620s	1.270s
sched_ft10 (10x10)	28139s/7292s	155.5s	191.0s	127.4s
sched_la16 (10x10)	21981s/4254s	121.7s	131.0s	104.7s

Table 3: Results on a set of hard job-shop scheduling problems

Problem	EGENET-FIX/RND	PL	PL + LF	PL + LLF
sched_la03 (10x5)	6.633s/9.300s	1.520s	1.370s	0.890s
sched_la04 (10x5)	13.47s/6.480s	1.350s	1.380s	0.930s
sched_ft10 (10x10)	18101s/2879s	109.7s	126.3s	112.6s
sched_la16 (10x10)	16027.2s/2380s	69.18s	70.76s	70.36s

Table 4: Results on a set of hard job-shop scheduling problems with bounds propagation preprocessing

value for a variable. MCHMGA relies more on the descent_single_pt_mutate operator which only picks one random value for a variable at a time. Thus, it is less sensitive to the domain sizes of the variables. The comparison without using any look forward shows that full look forward does not seem to be very beneficial, while lazy look forward can definitely be worthwhile. For the bounds propagated examples, lazy look forward does not provide as much benefit as in the original problems.

The versions of MCHMGA perform 10 to 100 times faster than the EGENET solvers showing the advantages of applying the population-based learning and look-forward algorithm on these hard CSPs. The MCHMGA improved with the population-based learning, lazy look-forward algorithm and bounds propagation technique performs the best among the improved MGAs demonstrating the efficiency of this combination on solving hard CSPs. This can be explained by the fact that whenever the pivot gene satisfies the constraints locally, the lazy look-forward algorithm can actively and carefully look into the other possible values for the pivot gene before invoking the generalized population-based learning mechanism to avoid the current local minima in the future search.

5 Concluding Remarks

CSPs can be solved by numerous approaches. Local search methods such as artificial neural networks [1, 8], evolutionary algorithms [2] or simulated annealing [5] have been successfully applied to such problems. Other local search approaches can be applied by transforming the CSP into a SAT problem which can then solved by a satisfiability algorithms such as GSAT [16] or DLM [17]. In this paper we focus on the use of evolutionary algorithms to solve CSPs.

Most conventional evolutionary algorithms, in general, perform "random walk" by genetic operators such as mutation or crossover, and depend on a selection function to guide the intrinsically parallel search of the different chromosomes in the population. However, in many moderately or highly constrained CSPs, this approach can be inefficient. To solve CSPs more efficiently with an evolutionary algorithm, we propose the integration into a micro-genetic algorithm of the min-conflict heuristic together with a learning mechanism to avoid earlier explored local minima. In this approach the search is mainly guided by the heuristic operators themselves instead of the selection function. In the other word, the heuristic operators are integrated more deeply into the evolutionary algorithms.

To demonstrate the feasibility of our search framework, we define a generic micro-genetic algorithm using a min-conflict mutation operator to improve the fitness values of the chromosomes in the population. We improved this algorithm with two novel heuristics: population-based learning and a look-forward algorithm. The resulting algorithm is able to solve moderately to highly constrained CSPs more efficiently.

There are several directions for further investigation. First, the applications of our algorithm on more real-life CSPs should be interesting. There are other variations and heuristics that may be advantageous to add to the micro-genetic algorithm. Stuckey and Tam [18] proposed lazy consistency as the most suitable technique for the local search methods to prune the search space, and showed how it could be used to improve the EGENET solver. Hence, the integration of lazy consistency in our generic algorithm is also interesting. Lastly, it would be interesting to study how to apply our algorithm in solving constrained optimization problems.

Acknowledgements

We wish to thank Edward Tsang for his useful comments. This work is partially supported by ARC Large Grant A49702580.

References

[1] A. Davenport, E.P.K. Tsang, C.J. Wang, and K. Zhu. GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of AAAI'94*, pages 325 – 330, 1994.

[2] Bowen James and Gerry Dozier. Solving Constraint Satisfaction Problems Using A Genetic/Systematic Search Hybrid That Realizes When to Quit. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 122–129, 1995.

[3] C. Wang and E. Tsang. Solving satisfaction problems using neural-networks. In *Proceedings of IEEE Second International Conference on Artificial Neural Networks*, pages 295 – 299, 1991.

[4] K.M.F. Choi. A lagrangian reconstruction of a class of local search methods. Master's thesis, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong, 1998.

[5] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; Part II, graph coloring and number partitioning. *Operations Research*, 39(3):378 – 406, 1991.

[6] G. Dozier, J. Bowen, and D. Bahler. Solving small and large scale constraint satisfaction problems using a heuristic-based microgenetic algorithm. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 306–311, 1994.

[7] H. Adorf and M. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Neural Networks*, 1990.

[8] J.H.M. Lee, H.F. Leung, and H.W. Won. Extending GENET for Non-Binary CSP's. In *Proceedings of Seventh International Conference on Tools with Artificial Intelligence*, pages 338–343, 1995.

[9] J.H.M. Lee, H.F. Leung, and H.W. Won. Towards a more efficient stochastic constraint solver. In *Proceedings of Principles and Practice of Constraint Programming (CP96)*, pages 338–352, 1996.

[10] Charles L. Karr. Air-injected hydrocyclone optimization via genetic algorithm. In *Handbook of Genetic Algorithms*.

[11] K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[12] Paul Morris. Solutions without exhaustive search: An iterative descent method for binary constraint satisfaction problems. In *AAAI-90 Workshop On Constraint Directed Reasoning Working Notes*.

[13] Marı́a Cristina Riff Rojas. From Quasi-Solutions to Solution: An Evolutionary Algorithm to Solve CSP. In *Proceedings of Principles and Practice of Constraint Programming (CP96)*, pages 367 – 381, 1996.

[14] S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161 – 205, 1992.

[15] Norman Sadeh. A set of hard job-shop scheduling problems. Available by anonymous FTP from "cimds3.cimds.ri.cmu.edu".

[16] B. Selman, H. Lavesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *AAAI-92*, pages 440–446, 1992.

[17] Y. Shang and B. W. Wah. A discrete lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–100, 1998.

[18] Peter Stuckey and Vincent Tam. Extending egenet with lazy constraint consistency. In *Proceeding of the IEEE Ninth Int. Conf. on Tools with Artificial Intelligence*, pages 248–259, 1997.

[19] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.