SOLVING THE RADIO LINK FREQUENCY ASSIGNMENT PROBLEM WITH THE GUIDED GENETIC ALGORITHM

T. L. LAU, and E. P. K. TSANG

University of Essex
Dept. of Computer Science
Wivenhoe Park
Colchester CO4 3SQ
United Kingdom
email: {tllau, edward}@essex.ac.uk

Received March 27, 1998
Revised March 27, 1998

The Radio Link Frequency Assignment Problem is an abstraction of a real life military application that involves the assigning of frequencies to radio links. This problem set consists of eleven instances that are classed as either a Constraint Satisfaction Optimization Problem or a Partial Constraint Satisfaction Problem. Each problem has different optimization and constraint requirements, and can have up to 916 variables, and up to 5548 constraints.

The Guided Genetic Algorithm (GGA) is a hybrid of Genetic Algorithm and meta-heuristic search algorithm Guided Local Search. As the search progresses, GGA modifies both the fitness function and fitness template of candidate solutions based on feedback from constraints. In this paper, we have shown that GGA has the best optimality-robustness advantage over current published results.

Keywords: Genetic Algorithm, Constraint Satisfaction Optimization Problem, Partial Constraint Satisfaction Problem

1 Introduction

A finite Constraint Satisfaction Problem (CSP) can be described as a problem with a finite set of variables, where each variable is associated with a finite domain. Relationships between variables constraint the possible instantiations they can take at the same time [1, 2]. To solve a CSP, one must find the solution tuple that instantiate variables with values of their respective domains, and that these instantiations do not violate any of the constraints. Our area of research is in Constraint Satisfaction Optimization Problem (CSOP) and Partial Constraint Satisfaction Problem (PCSP), two variations of the CSP.

In the realms of CSP, the instantiation of a variable with a value from its domain is called a *label*. A simultaneous instantiation of a set of variables is called a *compound label*, which is a set of labels. A *complete compound label* is one that assigns values, from the respective domains, to all the variables in the CSP.

A CSOP is a CSP with an objective function f that maps every complete compound label to a numerical value. The goal is to find a complete compound label S such that $f(S)$ gives an optimal value, and that no constraint is violated. A PCSP is similar to a CSOP except that the complete compound label may have variable instantiations that violate some of its constraints. Violation is unavoidable because the constraints are so tight that a satisfiable solution does not exist, or cannot be found [3, 1]. Deciding which constraint to violate is influenced by its cost and type. Hard constraints are types of constraints that must not be violated, whereas soft constraints may. The sum cost of all violated constraints is reflected in the objective function, further to other optimization criteria.

1.1 The Radio Link Frequency Assignment Problem

1.1.1 Background

The EUCLID CALMA (Combinatorial Algorithms for Military Applications) consortium is a group of six research bodies in Europe that was formed to investigate the use of AI techniques to aid military decisions. The Radio Link Frequency Assignment Problem (RLFAP) is a case study proposed within the group to observe the effectiveness of different approaches. It is an abstraction of a real world military application that is concerned with assigning frequencies to radio links. The RLFAP[1] contains eleven instances with various optimization criteria, made publicly available through the efforts of the French Centre d'Electronique l'Armament. RLFAP is NP-hard and is a variation on the T-graph colouring problem introduced in [4].

1.1.2 Types of Instances

Each instance in the RLFAP has a set of files that describe its variables, their domains, the constraints, and the objective. In addition, we are also given information on the respective optimization requirements based on the solubility of the problem. Optimization criteria describe the interpretation of variable instantiations and the means of measuring their desirability; thus shaping the objective function of our search routine, whereas the solubility of the problem states if a solution can be found under the condition that

[1] RLFAP is available at the Centre d'Electronique l'Armament (France), via ftp at ftp.cert.fr/pub/bourret.

no constraint was violated. If an instance can be solved without constraint violation, then its optimality is defined as either O1 or O2, otherwise it is O3 (see below). For an insoluble instance, we use the violation cost of each constraint to solve the instance as a PCSP. In this paper, regardless of the problem's solubility, all instances in RLFAP are solved as PCSP since a CSOP problem can be mapped into a PCSP by giving each constraint a violation cost. This violation cost is the same value for all constraints in the instance.

O1 - optimal solution is one with the fewest number of different values in its variables.

O2 - optimal solution is one where the largest assigned value is minimal.

O3 - if a the problem cannot be solved without violating constraints, find a solution that minimizes the objective function as follows:

$$\begin{aligned} & a_1 \times nc_1 + a_2 \times nc_2 + a_3 \times nc_3 + a_4 \times nc_4 + \\ & + b_1 \times nv_1 + b_2 \times nv_2 + b_3 \times nv_3 + b_4 \times nv_4 \end{aligned} \tag{1}$$

Where nc_i is the number of violated constraints of priority i, nv_i is the number of modified variables with mobility i. Mobility for a radio link states the cost for changing the frequency from its assigned default. The values of the weights a_i and b_i are given if necessary.

All constraints in the RLFAP are binary; that is, each constraint operates on the values in two variables. These constraints test the absolute difference of two variables in a candidate solution, where this logical test can belong to either of the two following classes:

C1 - the absolute difference must be lesser than a constant.

C2 - the absolute difference must be equal to a constant.

Table 1 lists the instances, their characteristics and its objective. From this table, we can observe that the RLFAP contains instances that are varied in both the optimization and constraint criteria. Further, the number of variables, their domain sizes, and the number of constraints on these variables make the RLFAP a non-trivial problem set for any algorithm. The RLFAP would not only test the quality and robustness of an algorithm, but also its flexibility to adapt to the different optimization and constraint criteria of each instance.

Table 1: Characteristics of RLFAP instances.

Instance	No. of variables	No. of constraints	Souble	Minimize	Type
scen01	916	5548	Yes	Number of different values used	O1
scen02	200	1235	Yes	Number of different values used	O1
scen03	400	2760	Yes	Number of different values used	O1
scen04	680	3968	Yes	Number of different values used	O1
scen05	400	2598	Yes	Number of different values used	O1
scen06	200	1322	No	The maximum value used	O2
scen07	400	2865	No	Weighted constraint violations	O3
scen08	916	2744	No	Weighted constraint violations	O3
scen09	680	4103	No	Weighted constraint violations and mobility costs	O3
scen10	680	4103	No	Weighted constraint violations and mobility costs	O3
scen11	680	4103	Yes	Number of different values used	O1

2 The RLFAP in PCSP Expression

A PCSP is defined as a quadruple of $\{Z, D, C, f\}$ where Z is a finite set of variables. With respect to Z, D is a function that maps every variable to a set of values, which is called a domain. C is a finite set of constraints that affect a subset of the variables, and each constraint has a cost for its violation. The objective function f returns a magnitude based on the instantiation of the variables and the satisfaction of constraints. In the RLFAP, each instance has a set of files that conveniently describe the respective Z, D and C sets.

2.1 Variables and Domains

For any of the RLFAP instance with m variables, let q_j be a variable in Z representing one radio link. For each variable q_j in Z, there is one associated domain mapped by the function D, denoted by $D(q_j)$, which contains a set of n values, each value representing a valid frequency that can be assigned to the variable.

$$Z = \{q_1, q_2, \ldots, q_m\} \tag{2}$$

$$where\ \forall q_j \in Z: \quad D(q_j) = \{freq_1, freq_2, \ldots, freq_n\} \tag{3}$$

2.2 Constraints

The constraint set C consists of n elements, representing n constraints in the instance. Each element in C consist of the constraint c_i and its cost $cost_i$[2] (Eq. 4). As discussed in section 1.1.2, there are two types of binary constraints in the RLFAP; C1 and C2 which we formulate into Eq. 5. In that equation, q_a and q_b are two variables from a candidate solution, and z is a constant. Eq. 6 states that constraint c_i returns a binary value that is 1 for a violation and 0 otherwise.

$$C = \{< c_1, cost_1 >, < c_2, cost_2 >, \ldots, < c_n, cost_n >\} \quad (4)$$

$$\forall c_i \in C : \begin{cases} c_i \equiv |q_a \quad q_b| < z, \text{if } c_i \text{ is type C1} \\ c_i \equiv |q_a \quad q_b| = z, \text{if } c_i \text{ is type C2} \end{cases} \quad (5)$$

$$where\ c_i = \begin{cases} 1, \text{constraint is violated} \\ 0, \text{otherwise} \end{cases} \quad (6)$$

2.3 Objective Function

The respective objective function f of the instances in RLFAP are stated in Table 1. The objective functions are also explained in section 1.1.2.

3 Algorithms

CSPs and CSOPs are generally NP-hard [1] and although heuristics have been found useful in solving them, most systematic search algorithms are deterministic and constructive [5], and would thereby be limited by the combinatorial explosion problem. Systematic methods include search and inference techniques. These search methods are complete, so they are able to guarantee a solution, or to prove that one does not exist. Thus systematic techniques will, if necessary, search the entire problem space for the solution [6].

The combinatorial explosion is an obstacle faced by systematic search methods for solving realistic CSPs, and in looking for optimal and or near-optimal solutions in CSOPs. In optimization, to ensure that the solution found is the optimal, systematic search algorithms would need to exhaust the entire problem space to establish that fact.

Stochastic search methods are normally incomplete. They are not able to guarantee that a solution can be found, and neither can they prove that a solution does not exist. They forgo completeness for efficiency. Often, stochastic search methods can be faster in solving CSOPs than systematic

[2]Cost for violating the constraint.

methods [7]. Many publications such as [8, 9, 10] demonstrated on several large problems that systematic search algorithms fail to solve, but stochastic alternatives efficiently conquer.

3.1 Genetic Algorithms

Geneitc Algorithms are stochastic search algorithms that borrow some concepts from nature [11, 12, 13]. GA maintains a *population pool* γ of candidate solutions called *strings* or *chromosomes*. Each chromosome γ_p is a collection of α building blocks known as *genes*, which are instantiated with values from a finite domain. Let $\gamma_{p,q}$ denote the value of gene q in chromosome p in the population γ.

Associated with each chromosome is a *fitness* value which is determined by a user defined function. The function returns a magnitude that is proportional to the candidate solution's suitably and/or optimality. Fig. 1 shows the control and data flow of a canonical GA. At the start of the algorithm, an initial population is generated. Initial members of the population may be randomly generated, or generated according to some rules. The *reproduction operator* selects chromosomes from the population to be parents for a new chromosome and enters them into the *mating pool*. Selection of a chromosome for parenthood can range from a totally random process to one that is baised by the chromosome's fitness.

The *cross-over operator* oversees the mating process of two chromosomes. Two parent chromosomes are selected from the mating pool randomly and the *cross-over rate*, which is a real number between zero and one, determines the probability of producing a new chromosome from the parents. If the mating was performed, a child chromosome is created which inherits complementing genetic material from its parents. The cross-over operator decides what genetic material from each parent is passed onto the child chromosome. The new chromosome produced is entered into the *offspring pool*.

The *mutation operator* takes each chromosome in the offsrping pool and randomly change part of its genetic make-up, ie. it's content. The probability of mutation occuring on any chromosome is determined by the user specified mutation rate. Chromosomes, mutated or otherwise, are put back into the offspring pool after the mutation process.

Thus each new generation of chromosomes are formed by the action of genetic operators (reproduction, cross-over and mutation) on the older population. Finally, the members of the population pool are compared with those of the offspring pool. The chromosomes are compared via their fitness value to derive a new population, where the weaker chromosomes may be eliminated. In exact, weaker members in the population pool is replaced by the fitter child chromosomes from the offspring pool. The heuristic for

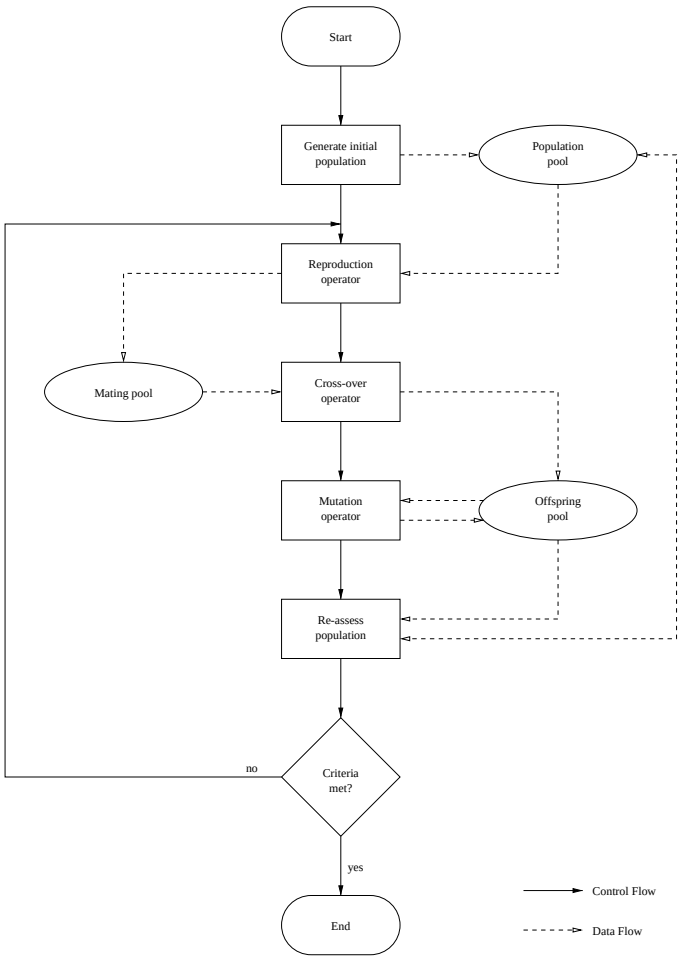

Figure 1: A canonical Genetic Algorithm

assessing the survival of each chromosome into the next generation is called the *replacement strategy*.

The process of reproduction, cross-over, mutation and formation of a new poplution completes one *generation* cycle. A GA is left to progress through generations, until certain criteria (such as a fixed number of generations, or a time limit) are met. GAs were initially used for machine learning systems, but it was soon realised that GAs have great potential in function optimization [14, 12, 11].

3.1.1 Shortcomings of GAs when solving CSPs

In applying the canonical GA to solve instances belonging to the CSP class, the problem of *high epistasis* often limits its success. Epistasis is the interaction between different genes in a chromosome. A candidate solution to a typical CSOP is often represented as a chromosome, where each gene in the chromosome describes a variable in the CSOP. Constraints influence both the values that sets of genes can take simulanteously and the overall fitness of that chromosome. Goldberg suggested high epistasis as an explanation to GAs failure in certain tasks [12].

3.2 Guided Genetic Algorithm

3.2.1 Background

Among our eariler work on CSOPs, we looked at the Processor Configuration Problem (PCP) [15, 16]. Briefly, the PCP is a real life CSOP where the task is to link up a finite set of processors into a network, whilst minimizing the maximum distance between these processors. Since each processor has a limited number of communication channels, a carefully planned layout will help reduce the overhead for message switching.

We developed a GA called the Lower Power Genetic Algorithm (LPGA) [17, 18] specifically for solving the PCP. LPGA is a two-phase GA approach where in the first phase, we run LPGA until a local optimum has been determined. The best chromosome from this run is analysed and used to construct a fitness template for use in the next phase. This fitness template is a map that defines undesirable genes, so influencing LPGA to change their contents. By insisting that crucial genes do not change, the evolution in the second phase shifts focus onto other parts of the string; resulting in a more compact search space.

LPGA found solutions better than results published so far in the PCP. It's success could be attributed to the use of an effective data representation and more importantly, the presence of an application specific penalty algorithm. In our effort to generalize LPGA, we sought to develop a GA that utilizes a dynamic fitness template constructed by a general penalty

algorithm. The Guided Genetic Algorithm (GGA) repeated in this paper was the result of this effort.

3.2.2 Overview of GGA

In our jounrey to develop GGA, we have taken liberty with some of the traditional GA concepts (such as the addition of a penalty operator, and an alternate interpretation of the mutation rate). These will be introduced as we progress through the rest of this paper. Comparing GGA in Fig. 2 against the canonical GA in Fig. 1, we could see the additions of a data collection called the fitness templates, a penatly operator (see 3.2.3) and a condition to activate that operator. Also added to Fig. 2 are the interactions between the penalty operator and the data space in GGA. Appended at the end of this section (section 3) are two tables (Table 2 and 3), summarizing for the readers' convenience, the terms and technology introduced henceforth.

The control flow of the GGA is very much similar to that of the canonical GA, described in section 3.1. After the start of the algorithm, an initial population is created. A new generation of chromosomes are derived from the parent chromosomes through the actions of the reproduction, cross-over and mutation operators. Both the cross-over and mutation operators (or any operator thereof) may be adapted to use the information provided by the penalty operator, via the *fitness template* of each chromosome which is collected in the *fitness templates* (explained in section 3.2.4). Memberships to the population pool are re-assessed by comparing the fitness of the chromosomes from the population and offspring pool. For the RLFAP, GGA was configured to use an *elitist replacement strategy*. Under this strategy, chromosomes from both the population and offspring pool are ranked by their fitness. The fittest n chromosomes in the ranking are used to form the next generation's population pool. In GGA, n is set to the size of the population pool.

New elements of the GGA comes into play at this point. The new population is surveyed for the possibility of being trapped in a local optimum. We can observe that when a search is trapped in a local optimum, it repeatedly returns the same solution since the neighbouring states does not offer any improvement. If the population is indeed trapped in a local optimum, the penalty operator is called. The penalty operator looks for undesirable features in the chromosomes and update the fitness template (or fitness templates), so that mutation and cross-over operators might fade out these features in the coming generations.

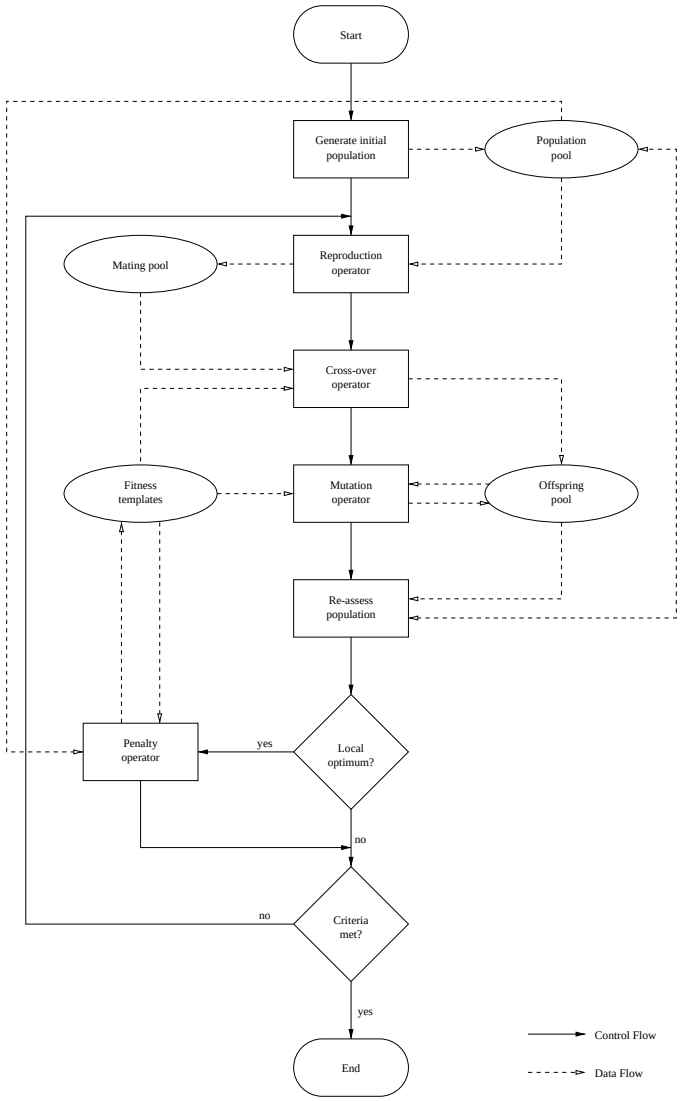

Figure 2: The Guided Genetic Algorithm

3.2.3 Penalty Operator

In LPGA, its use of a fitness template (generated by a specialized penalty algorithm) was the motivating force in the development of GGA. In the quest for a general penalty algorithm, we looked for functional similarities to LPGA and more importantly, that the nature of the penalty algorithm will not be obstructive to the operation of a canonical GA. The Guided Local Search (GLS) [19] developed by our research group is an intelligent search scheme for combinatorial optimization problems. It met our criteria and further, its conceptual simplicity and proven effectiveness in a range of well known problems was an added attraction [20, 21, 22, 23]. In GGA, we adapted GLS in the form of the *penalty operator*.

Solutions are characterized by a set of solution features θ, where a solution feature θ_i can be any property exhibited by the solution (Eq. 7). This property must be non-trival, such that it does not appear in all candidate solutions. Research on GLS has indicated that feature definition is not difficult, since the domain often suggests features that one could use. The application in this paper supports this point.

$$\theta = \{\theta_1, \theta_2, \ldots, \theta_m\} \tag{7}$$

In GGA, a fearure is limited to variable assignments (in GLS, it is more general); a feature in a chromosome may be exhibited by the simultaneous assignments of a group of genes. Thus the feature θ_i defines a set of positions in the chromosome representation. And the feature θ_i is represented by an *indicator function* τ_i in Eq. 8, which test the existence of that feature. For each feature θ_i, there is a *cost* η_i which rates that feature's presence in a solution in degrees of undesirability. Indicator functions and costs are application dependent, and so they are defined by the user.

$$\tau_i(\gamma_p) = \begin{cases} 1, & \text{solution } \gamma_p \text{ exhibits feature } \theta_i \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

Penalty counter ζ_i is a variable maintained by GGA that gives the degree of extent that the feature θ_i is penalized as the search is progressing; the counter is initialized to zero at the beginning of the search. A new fitness function called the *augmented cost function* g (Eq. 9) is used in place of function f, so that changes in penalty counters will affect the survival of chromosomes. The *regularization parameter* λ (adopted from GLS) measures the impact penalties have, with respect to function f.

$$g(\gamma_p) = f(\gamma_p) + \lambda \cdot \sum(\zeta_i \cdot \tau_i(\gamma_p)) \tag{9}$$

In GGA, if the fitness of the best chromosome remains unchanged for a specific number of generations, we conclude that it is trapped in a local optimum. The penalty operator comes in to analyze the best chromosome γ_p of the population for features to penalize. Penalties are used in GGA to guide the search to escape local optima. To evaluate the utility of penalizing individual features exhibited by a candidate solution, GGA (following GLS) takes into consideration the cost as well as the penalty counter (Eq. 10). Thus, for all features θ_i in the fittest chromosome γ_p that maximizes the function $util(\gamma_p, \theta_i)$ (Eq. 10), the related penalty counter ζ_i is incremented by one. It is hoped that by penalizing undesirable features, we can escape from the local optimum and suppress the occurance of these features in the coming generations.

$$util(\gamma_p, \theta_i) = \tau_i(s) \cdot \frac{\eta_i}{1 + \zeta_i} \tag{10}$$

3.2.4 Fitness Template

Central to the theme in GGA is the fitness templates. Besides the fitness function, the fitness templates offer an added channel of communication between the penalty operator, and the mutation and cross-over operators. The fitness template is a map that defines which genes in a chromosome are more susceptible to be changed during cross-over or mutation.

In GGA, each chromosome γ_p in the population is associated with exactly one *fitness template* δ_p. Each fitness template is made up of smaller units known as *weights* $\delta_{p,q}$, each of which corresponds to a gene $\gamma_{p,q}$. A weight $\delta_{p,q}$ is a positive integer. The "heavier" a gene appears (compared to its comrades), the greater are its chances of having its content altered. Therefore in the case of mutation, the weight of a gene is proportional to the probability that mutation may occur on it, relative to the weights of other genes in the same chromosome. This is especially useful when the number of genes in a chromosome is large, where random selection of genes might not be helpful. More details on the role of the fitness template with the mutation and cross-over operator will be given in their respective sections.

Weights in the fitness template for each chromosome are computed when the chromosome was first created, and after the penalty operator has penalized feature(s). Computation of weights are needed after these events because the content of either the chromosomes or the penalty counters have changed.

For a chromosome, the distribution process (Fig. 3) starts by initializing all weights to zero. It will check the chromosome for the presence of any features from the set θ. For a feature θ_i that exist in the chromosome[3], all

[3]Feature θ_i is present when its indicator function τ_i returns a one.

the weights related to the gene positions defined by θ_i is incremented with the value in its penalty counter ζ_i.

```
FUNCTION DistributePenalty( chromosome γ_p )
{
    FOR EACH weight δ_{p,q} RELATED TO chromosome γ_p
    {
        δ_{p,q} ← 0
    }

    FOR EACH solution feature θ_i IN feature set θ
    {
        IF τ_i(γ_p) = 1 THEN
        {
            FOR EACH gene position q defined by θ_i
            {
                δ_{p,q} ← δ_{p,q} + ζ_i
            }
        }
    }
}
```

Figure 3: Algorithm for the Distribution of Penalty

3.2.5 Cross-over Operator

The action of mating two individuals from the population produces a new child. Each parent contributes a set of genes which the child inherits. In GA, the process of choosing parents, deciding their respective contribution rights of genetic material, and the forging of a child chromosome from these material is the responsibility of the cross-over operator. The probabilty of cross-over occuring is controlled by the parameter *cross-over rate*. By assembling a new chromosome that contains parts of two parent chromosomes, it may introduce to the popultion a new point in the search space. And since the parents chosen for mating are selected with bias to their fitness, we hope that the child chromosome may be fitter.

Cross-over operators differ primarily from each other in the way that they choose the genes from the parents to form the child. In the canonical GA, one of the simplest form of cross-over is the *one-point cross-over* [12, 11]. In Fig. 4, we have two parent chromosomes whose genes are binary encoded. One random point along the length of the chromosomes are selected as the

cross-over point. Each parent donates one different part of their chromosome (defined by the cross-over point) to create the child chromosome.

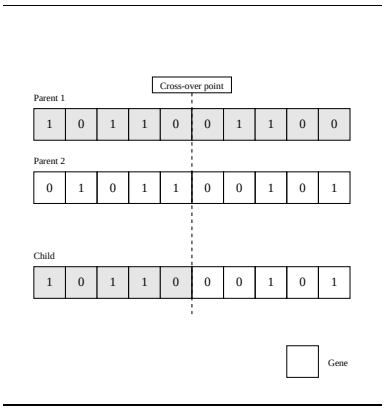

Figure 4: An example of the One-point Cross-over Operator in action

In GGA, we have adapted the cross-over operator to take advantage of the fitness template. Two chromosomes γ_p and $\gamma_{p'}$ are selected as parents to produce the child $\gamma_{p''}$. Each gene in chromosome γ_p competes against the corresponding gene in $\gamma_{p'}$ for a place in the child. This compeitition is a weighted random selection, influenced by the weights $\delta_{p,q}$ and $\delta_{p',q}$ of the respective genes; thus the "lighter" gene will have a greater chance to propagate its information to the child. Note that the child does not inherit the weights from its respective parent, since the child may represent a different solution from its parents, and thus requiring the penalty operator to re-assess it. The algorithm of GGA's cross-over operator is shown in Fig. 5, and Fig. 6 shows its effect when applied to the situation for one-point cross-over in Fig. 4.

The operator starts off by receiving two parents γ_p and $\gamma_{p'}$ from the mating pool. For each set of corresponding genes $\gamma_{p,q}$ and $\gamma_{p',q}$ in the parents, it computes the *sum* of their weights. The selection of the gene is randomly biased, such that the probability for either $\gamma_{p,q}$ or $\gamma_{p',q}$ to have its gene passed onto their child is $\frac{\gamma_{p',q}}{sum}$ and $\frac{\gamma_{p,q}}{sum}$ respectively; giving the advantage to a "lighter" gene, which we would want the child $\gamma_{p'',q}$ to inherit. This gene selection process is repeated for all genes in the parents. When a child chromosome is complete, its fitness and weights are computed.

3.2.6 Mutation Operator

Mutation produces variations in the population through altering the information that genes carry. The *mutation rate* states the probability that

```
FUNCTION CrossOver( parent chromosomes γ_p and γ_p' )
{
    FOR EACH gene position q IN the chromosome
    {
        sum ← δ_{p,q} + δ_{p',q}
        point ← random integer from {0, ..., sum    1}

        IF point < δ_{p,q} THEN
        {
            gene ← γ_{p',q}
        }
        ELSE
        {
            gene ← γ_{p,q}
        }
    }

    RETURN gene as γ_{p'',q} for the offspring
}
```

Figure 5: Algorithm of the GGA Cross-over Operator

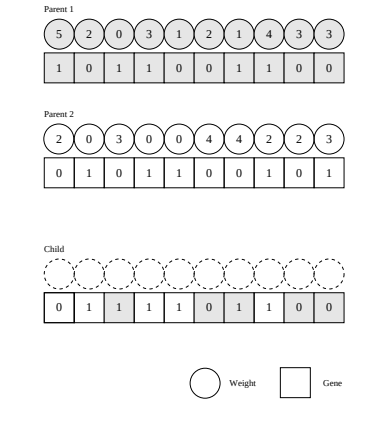

Figure 6: An example of the GGA Cross-over Operator in action
(Weights for the child is calculated afresh, not inherited)

mutation may occur on a chromosome. In GGA, the mutation rate is defined as a fraction of the size of each chromosome; the number of genes in a chromosome to mutate is the product of the mutation rate and the size of that chromosome.

In GGA, mutation (Fig. 7) acts on every child chromosome $\gamma_{p''}$ produced by the cross-over operator. For each chromosome, a number of genes are chosen (as above, decided by the mutation rate) to be modified. A gene $\gamma_{p'',q}$ is selected using the *roulette wheel selection method*. In this selection method, the probability for each gene to be picked is directly proportional to its weight. Thus a gene with a "heavier" weight (and therefore less desirable) compared to others in the chromosome, will have a greater chance of being selected. Appended below is a description of our implementation of the roulette wheel selection method.

> Given the chromosome $\gamma_{p''}$, we compute the sum of all the weights in the fitness template associated to this chromosome as $sum = \sum_{q=1}^{\alpha} \delta_{p'',q}$. The probabilty that a gene $\gamma_{p'',q}$ is selected is proportional to its weight over sum, ie. $P(\gamma_{p'',q}) = \frac{\delta_{p'',q}}{sum}$.

The next step for a selected gene $\gamma_{p'',q}$ is to seek an appropriate value for replacement. This could be totally random or in GGA's case, a value that will derive the best fitness (the biggest improvement) for the chromosome. In our algorithm, we have the variables *best* and *list*. The variable *best* holds the best fitness value, while *list* contains a list of values that will allow the chromosome to arrive at the fitness value in *best*. We step through all the values x_j in the domain $D(q)$ relevant to the gene $\gamma_{p'',q}$. If x_j produces a new fitness z greater[4] than *best*, *best* is set to z and *list* is emptied. However, if the z is equal to *best*, the value x_j is added to the *list*. When all values in the domain have been exhausted, we randomly instantiate $\gamma_{p'',q}$ with a value from *list*. Since at this point, *list* should contain all possible values that will give the chromosome $\gamma_{p''}$ the biggest improvement.

Updating of a gene's weight takes place after it's value has changed, where the weight associated with it is reduced by one unit [5] so that the probability of the same gene getting selected by the roulette wheel selector is reduced.

Gene mutation is repeated until the stopping criteria is met. As stated before, we stop mutating when the number of genes changed have reached a value that is the product of the mutation rate and the chromosome's length.

[4]Since RLFAP is a minimization problem, we would want the greatest decent.

[5]Since a weight is a positive integer, a weight will only be decremented if it is greater than zero.

```
FUNCTION Mutation( chromosome γ_{p''} )
{
    i ← 0

    WHILE i < mutation rate × α (length of chromosome)
    {
        q ← RouletteWheel(γ_{p''})
        best ← g(γ_{p''})
        list ← γ_{p'',q}

        FOR EACH value x_j IN domain D(q)
        {
            γ_{p'',q} ← x_j
            z ← g(γ_{p''})

            IF z ≥ best THEN
            {
                IF z > best THEN
                {
                    best ← z
                    list ← {}
                }

                list ← list + x_j
            }
        }

        γ_{p'',q} ← random value in list
        i ← i + 1
    }

    RETURN the mutated chromosome γ_{p''}
}
```

Figure 7: Algorithm of the GGA Mutation Operator

Table 2: Components of GGA

Algorithms	Purpose
Cross-over operator	Uses the fitness templates of two parent chromosomes to decide each parent's contribution of genetic material towards creating a child chromosome.
Mutation operator	The fitness template of a chromosome is used to guide in the alteration of the chromosome's genetic content.
Penalty operator	This operator detects and selects undesirable solution features in a chromosome to penalize. Penalization involves incrementing penalty counters of the assocaited features.
Local optimum detector	Detects if the search is trapped in a local optimum. If it is, the penalty operator is called.
Distribute Penalty	If a solution feature is present in a chromosome, the penalty counter associated with this feature is added onto the weights of the genes that are constituents of this feature.
Data structures	**Purpose**
Weight δ	Each gene has one weight. The weight is a measure of undesirability of the gene's current instantiation, compared to the rest of the chromosome.
Fitness template	A fitness template is a collection of weights. Each chromosome has one fitness template.
Solution feature θ	Solution features are domain specific and user defined. A feature is exhibited by a set of variable assignments that describes a non-trival property of a problem.
Penalty counter ζ	Each feature has one penalty counter. A penalty counter keeps count of the number of times its related solution feature has been penalized since the start of the search.

Table 3: Inputs and Parameters to GGA

Inputs/Parameters	Purpose
Solution feature θ	See Table 2
Cost η	Each solution feature has a cost to rate its undesirability of presence.
Indicator function τ	Each solution feature has a user defined indicator function that tests for the feature's presence in a chromosome.
Objective function	A function that maps each solution to a numerical value.
Regularization parameter λ	A parameter that determines the proportion of contribution that penalties have in an augmented fitness function.
Augmented fitness function g	A function that is the sum of the objective function on a chromosome and the penalties of features that exist in it.
Mutation rate	A fraction that defines the number of genes in the chromosome to mutate.
Cross-over rate	The probability that cross-over will occur between two chromosomes.

4 Preparing GGA to solve RLFAP

In section 2, we expressed the RLFAP as a formal PCSP. In this section, we discuss the steps needed to adapt those definitions into a form that GGA can use.

The feature set θ is a union of the feature set of constraints θ_{cst} and the set of mobility of radio links θ_{mbt} (Eq. 11). Constraint c_i defined in Eq. 5 is recast as a feature in the set θ_{cst} (Eq. 12), where a one is returned if the constraint cannot be satisfied, and zero otherwise. The value of cost ζ_{cost_i} to each constraint c_i depends on the nature of the instance. If the instance is soluble, then all ζ_{cst_i} are set to a large value; usually 10000, to signify that the constraint must not be borken (ie. hard constraints). For insoluble instances, ζ_{cst_i} is set to the weights given for its priority class (see section 1.1.2). Similar to soluble instances, hard constraints in insoluble instances will have their ζ_{cst_i} set to a large value. The set θ_{cst} has n features, where n is the number of constraints in the instance.

For the O3 objective type of instances, we need to minimize the mobility cost of our candidate solution, in addition to minimizing constraints violation costs. The set of mobility cost defines our next feature set, θ_{mbt} (Eq. 13). For each variable in these instances, there is a mobility cost ζ_{mbt_i} and a default assigned frequency $default_i$. If in our candidate solution, a variable has been assigned a value different from its default $default_i$, then a one is returned and zero otherwise. The mobility cost ζ_{mbt_i} is set to the weights given for its prioriy class (again see section 1.1.2). There are radio links whose frequency should never change, and the mobility cost for these have been set to a large value. The feature set θ_{mbt} has n features, where n is the number of variables in the instance.

$$\theta = \{\theta_{cst}, \theta_{mbt}\} \tag{11}$$

$$\forall \theta_i \in \theta_{cst} : \tau_{cst_i}(\gamma_p) \equiv \begin{cases} 1, \text{if C1 and } |\gamma_{p,a} \quad \gamma_{p,b}| \geq z_i \\ 1, \text{if C2 and } |\gamma_{p,a} \quad \gamma_{p,b}| \neq z_i \\ 0, \text{otherwise} \end{cases} \tag{12}$$

$$\forall \theta_i \in \theta_{mbt} : \tau_{mbt_i}(\gamma_p) \equiv \begin{cases} 1, & \text{if } \gamma_{p,i} \neq default_i \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

For all instances in the RLFAP, we seek to minimize the function g (Eq. 14). In g, the function f depends on the objective type of the instance (Eq. 15). The cost ζ_i and τ_i both refers to the unified feature set of θ. They will automatically associate with ζ_{cst_i} and τ_{cst_i}, or ζ_{mbt_i} and τ_{mbt_i} where applicable. Thus the value of n in Eq. 14 is the sum of number of features in θ_{cst} and θ_{mbt}.

$$g(\gamma_p) = f(\gamma_p) + \lambda \cdot \sum_{i=1}^{n} (\zeta_i \cdot \tau_i(\gamma_p)) \quad (14)$$

$$f(\gamma_p) = \begin{cases} \text{if O1,} & \text{number of different values used in } \gamma_p \\ \text{if O2,} & \text{largest value used in } \gamma_p \\ \text{if O3,} & a_1 \times nc_1 + a_2 \times nc_2 + a_3 \times nc_3 + a_4 \times nc_4 + \\ & + b_1 \times nv_1 + b_2 \times nv_2 + b_3 \times nv_3 + b_4 \times nv_4 \end{cases} \quad (15)$$

5 Benchmark

The RLFAP benchmark results by algorithms devised within the CALMA group was reported by Tiourine et al. in [24]. In this section, we compare GGA's results with the CALMA algorithms (section 5.2). Further, we will also evaluate the examine the value that GGA adds to the canonical GLS (section 5.3).

5.1 Test Environment

In our physical environment, GGA was written in C++ and complied using GNU GCC version 2.7.1.2. The code runs on an IBM PC compatible with a Pentium 133 MHz processor, 32MB of RAM and 512KB of Level 2 cache. Both compilation and executaion of GGA was performed on the Linux operating system, using kernel version 2.0.27. Under GGA's environment, we have a mutation rate and cross-over rate of 1.0, a population size of 20, and a stopping criterion of 100 generations. For the fitness function g, λ has a value of 10.

5.2 Comparing Quality of Solutions

In Table 4, we see the published results of all the CALMA algorithms, GLS and GGA. Algorithms from the CALMA project groups consist of either complete or stochastic methods. The results recorded in the table are from the best solution each algorithm had generated. For soluble instances (scen01, scen02, scen03, scen04, scen05 and scen11)[6], we report the number of frequencies above the known optimum that each solution (generated by the respective algorithms) uses. Results for the insoluble instances are reported as the percentage deviation from the best known reported solution.

For soluble instances, we observe that only seven out of the 13 algorithms[7] were able to provide a solution to all the instances. Of the six

[6] Instance scen06 was found to be insoluble, and thus solved as an O3 problem.

[7] Genetic Algorithms (LU) was designed for PCSP, and so it did not attempt any of the soluble instances.

soluble instances, GGA failed to return an optimum solution only for instance *scen11*. This instance has proved to be difficult for most of the algorithms, since only three algorithms (Taboo Search (EUT), Branch and Cut (DUT,EUT) and Constraint Satisfaction (LU)) were able to return a solution on par with the best known. Only two algorithms (Branch and Cut (DUT,EUT) and Constraint Satisfaction (LU) were able to report solutions that gave the most optimal resutls. However, these two algorithms were limited to solving soluble instances only.

Looking at insoluble instances, we see that 11 out of the 14 algorithms were able to solve these PCSP instances. Of the 11, nine of these algorithms managed to provide a satisfactory solution to the instances. Of note is the Genetic Algorithms (LU), which has the best known solutions.

Overall, we see that top performers in each category (soluble and insoluble) are limited to only that category; Branch and Cut (DUT,EUT) and Constraint Satisfaction (LU) for soluble instances, and Genetic Algorithms (LU) for insoluble instances. Of the total of 14 algorithms, 10 were applicable to both categories. Of these 10, only four algorithms were able to provide a satisfactory solution to all the 11 instances. Of the four, GLS and GGA consistently gave solution quality very close to the best known solutions.

5.3 Comparing GGA and GLS

From Table 4, we see that solution quality reported by GLS and GGA were very much the same except in soluble instance *scen11*, where GGA managed to better GLS's result. The amount of CPU time required for computing these results have shown GLS to be much superior. However, it is unfair to view GGA as just a parallel version of GLS. In section 3.2, we describe the mechanics of GGA. We have integrated GLS as a component of GGA; ie. as the penalty operator. The feedback from the GLS is used at two levels within GGA. On one level, GLS modifies the objective (fitness) function of GGA to influnce its search. On another level, information from GLS are encoded into the template of each chromosome, which rates the relative fitness of each gene in the chromosome. The templates are used to influence the cross-over and mutation processes. But because GGA manages several candidate solutions at the same time, it will have a more complex (and perhaps less effective) means to detect local optimum traps.

Besides solution quality and computation speed, one other measure is robustness. Robustness of an algorithm measures the consistency of the solutions it returns. In certain environments, one may need to be absolutely sure that the solution returned by an algorithm is, or very near the optimum. Following, we compare the statistics of GGA and GLS.

For the comparison, GGA runs with a population of only five chromo-

Table 4: Comparison of GGA with GLS and the CALMA project algorithms.

	Soluble Instances							Insoluble Instances						
Instance (scen)	01	02	03	04	05	11	Time	06	07	08	09	10	Time	Platform
Simulated Annealing (EUT)	2	0	2	0	0	2	1min	6%	65%	5%	0%	0%	310min	SUN Sparc 4
Taboo Search (EUT)	2	0	2	0	-	0	5min	-	-	-	-	-	-	SUN Sparc 4
Variable Depth Search (EUT)	2	0	2	0	-	10	6min	3%	0%	14%	0%	0%	85min	SUN Sparc 4
Simulated Annealing (CERT)	4	0	0	0	-	10	41min	42%	1299%	70%	2%	0%	42min	SUN Sparc 10
Tabu Search (KCL)	2	0	0	0	0	2	40min	167%	1804%	566%	8%	1%	111min	DEC Alpha
Extended GENET (KCL)	0	0	0	0	0	2	2min	12%	27%	40%	-	-	20min	DEC Alpha
Genetic Algoritms (UEA)	6	0	2	0	-	10	24min	0%	386%	134%	3%	0%	120min	DEC Alpha
Genetic Algorithms (LU)	-	-	-	-	-	-	-	0%	0%	0%	0%	0%	hours	DEC Alpha
Partial Constraint Satisfaction (CERT)	4	0	6	0	0	-	28min	83%	2563%	246%	47%	12%	6min	SUN Sparc 10
Potential Reduction (DUT)	0	0	2	0	0	-	3min	27%	-	-	4%	1%	10min	HP 9000/720
Branch and Cut (DUT,EUT)	0	0	0	0	0	0	<10min	-	-	-	-	-	-	-
Constraint Satisfaction (LU)	0	0	2	0	0	0	hours	-	-	-	-	-	-	PC
Guided Local Search (UE)	0	0	0	0	0	6	20sec	4%	9%	7%	0.7%	0.003%	2.88min	DEC Alpha
Guided Genetic Algorithm (UE)	0	0	0	0	0	2	30min	4%	9%	7%	0.7%	0.003%	2.88min	PC Linux
Best known solution	16	14	14	46	792	22		3437	343594	262	15571	31516		

Results for the soluble instances are reported as the number of frequencies more than the optimum used.
Results for the insoluble instances are reported as the percentage deviation from the best known solution.

CERT	Centre d'Etudes et de Recherces de Toulouse, France
DUT	Delft University of Technology, The Netherlands
EUT	Eidhoven Univeristy of Technology, The Netherlands
KCL	King's College London, United Kingdom
LU	Limburg University, Maastricht, The Netherlands
UEA	University of East Anglia, Norwich, United Kingdom
UE	University of Essex, United Kingdom

somes[8]. We introduce variation of GLS, called *GLS5* to compete with GGA on even grounds. GLS5 is five GLS running concurrently of each other, each maintaining its own candidate solution. Run time and iteration count for each GLS thread within GLS5 has also been extended to meet with GGA's. At the end of each run, only the best solution from GLS5 was used. For all 11 instances, the three algorithms (GGA, GLS and GLS5) were each sampled 50 solutions for each instance.

The results from Table 5 shows GGA to have better robustness than GLS or GLS5 in soluble instances. Whereas results for insoluble instances are mixed, with both algorithms having results very close to each other.

6 Conclusion

GGA was devised as a GA for arresting the effect of high epistatis when GAs are deployed to solve problems such as those from the CSP class. In the benchmarks, we have shown that GGA adds value to the canonical GLS. And that overall, GGA performed well against the other algorithms. As a GA, GGA was more flexible than Genetic Algorithms (LU), and performed better than Genetic Algorithms (UEA).

The integration of GLS and the introduction of new elements to the foundation of the canonical GA gave GGA a technique of measuring gene fitness for a chromosome, and the provision for multi-criteria optimization. By knowing a gene's fitness within a chromosome, one could understand the magnitude of its contribution to the overall fitness. Gene fitness influence the effects that genetic operators have on them, encouraging change to genes with low fitness, whilst protecting the healthy ones.

For most applications where the users are more concerned with turnaround time and less so on robustness, GLS is clearly the better choice. But for mission critical applications, or applications where time is not as tight, GGA can guarantee robustness and may perform better than GLS at times.

References

[1] E. P. K. Tsang, *Foundations of Constraints Satisfaction.* Academic Press Limited, 1993.

[2] E. Freuder and A. Mackworth, *Constraints-based Reasoning.* MIT Press, 1994.

[8] This is the minimum number of chromosome needed to maintain any advantage over GLS.

Table 5: Comparing Robustness between GGA, GLS and GLS5.

	Best Solution	Average Cost			Standard Deviation			Worst Solution		
Instance		GLS	GLS5	GGA	GLS	GLS5	GGA	GLS	GLS5	GGA
scen01	16	18.6	17.0	16.0	2.3	0.8	0.0	22	18	16
scen02	14	14.0	14.0	14.0	0.0	0.0	0.0	14	14	14
scen03	14	15.4	14.4	14.0	1.3	0.4	0.0	18	16	14
scen04	46	46.0	46.0	46.0	0.0	0.0	0.0	46	46	46
scen05	792	792.0	792.0	792.0	0.0	0.0	0.0	792	792	792
scen06	3628	4333.8	4029.6	4029.6	766.0	538.2	529.3	6042	6028	6028
scen07	427054	530641.1	510532.5	51344.8	79666.7	75149.4	75612.8	700685	694011	698837
scen08	294	335.7	322.6	320.5	34.7	23.1	21.5	377	372	368
scen09	15805	15999.7	15895.0	15889.6	194.7	112.6	109.3	16340	16280	16124
scen10	31533	31686.6	31621.4	31626.1	146.1	101.7	104.8	31942	31922	31922
scen11*	28	-	33.9	30.2	-	3.1	1.7	-	36	32

* For *scen11*, results for GLS could not be computed because it did not return a satisfiable solution for some runs.

[3] E. Freuder and R. Wallace, "Partial constraint satisfaction," *Artificial Intelligence*, vol. 58, pp. 21–70, 1992.

[4] K. Hale, "Frequency assignment: Theory and applications," in *IEEE*, vol. 68, pp. 1497–1514, 1980.

[5] P. Meseguer, "Constraint satisfaction problems: An overview," *AI Communications*, vol. 2, pp. 3–17, Mar. 1989.

[6] Kumar, "Algorithms for constraint satisfaction problems: A survey," *AI Magazine*, vol. 13, no. 1, pp. 32–44, 1992.

[7] E. Freuder, R. Dechter, B. Selman, M. Ginsberg, and E. Tsang, "Systematic versus stochastic constraint satifaction," in *14th International Joint Conference on Artificial Intelligence*, 1995.

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

[9] B. Selman, H. Levesque, and D. Mitchell, "A new method for solving hard satisfiability problems," in *10th National Conference on Artificial Intelligence*, pp. 440–446, 1992.

[10] P. Langley, "Systematic and non-systematic search strategies," in *Artificial Intelligence Planning Systems: Proceedings of the first international conference*, pp. 145–152, 1992.

[11] L. Davis, *Handbook of Genetic Algorithm.* Von Nostrand Reinhold, 1991.

[12] D. E. Goldberg, *Genetic Algorithm in Search, Optimization and Machine Learning.* Addison-Wesley Pub. Co., Inc., 1989.

[13] J. Holland, "Some practical aspects of adaptive systems theory," *Electronic Information Handling*, pp. 209 – 217, 1965.

[14] A. Bethke, "Genetic algorithms as function optimizers," Tech. Rep. 197, Logic of Computer Group, University of Michigan, USA, 1978.

[15] A. Chalmers and S. Gregory, "Constructing minimum path configurations for multiprocessor systems," Tech. Rep. CSTR-92-12, Computer Science Department, University of Bristol, UK, Apr. 1992.

[16] T. Warwick, *A GA Approach to Constraint Satisfaction Problems.* PhD thesis, Department of Computer Science, University of Essex, UK, 1995.

[17] T. L. Lau and E. P. K. Tsang, "Applying a mutation-based genetic algorithm to the processor configuration problem," in *IEEE 8th International Conference on Tools with Artificial Intelligence*, 1996.

[18] T. L. Lau and E. P. K. Tsang, "Solving the processor configuration problem with a mutation-based genetic algorithm," *International Journal on Artificial Intelligence Tools*, vol. 6, no. 4, 1997.

[19] C. Voudouris, *Guided Local Search*. PhD thesis, Department of Computer Science, University of Essex, UK, 1997.

[20] E. P. K. Tsang and C. Voudouris, "Fast local search and guided local search and their application to british telecom's workforce scheduling problem," *Operations Research Letters*, vol. 20, pp. 119–127, Mar. 1997.

[21] C. Voudouris and E. P. K. Tsang, "Guided local search," Tech. Rep. CSM-247, Department of Computer Science, University of Essex, UK, Aug. 1995.

[22] C. Voudouris and E. P. K. Tsang, "Function optimization using guided local search," Tech. Rep. CSM-249, Department of Computer Science, University of Essex, UK, Sept. 1995.

[23] C. Voudouris and E. P. K. Tsang, "Partial constraint satisfaction problems and guided local search," in *Practical Application of Constraint Technology*, pp. 337–356, Apr. 1996.

[24] S. Tiourine, C. Hurkens, and J. Lenstra, "An overview of algorithmic approaches to frequency assignment problems," in *CALMA Symposium, Scheveningen*, 1995.