Warwick, T. & Tsang, E.P.K., Tackling car sequencing problems using a generic genetic algorithm, Evolutionary Computation, MIT Press, Vol.3, No.3, 1995, p267-298

Tackling Car Sequencing Problems Using a Generic Genetic Algorithm

Terry Warwick & Edward P K Tsang
Department of Computer Science
University of Essex
Colchester CO4 3SQ
United Kingdom
email: {warwt, edward}@essex.ac.uk
A revised version to appear in Evolutionary Computation, MIT Press

Key words: genetic algorithms, constraint satisfaction, car sequencing

Abstract

The car sequencing problem (CarSP) was seen as a challenge to artificial intelligence. The CarSP is a version of the *job-shop scheduling problem* which is known to be NP-complete. The task in the CarSP is to schedule a given number of cars (of different types) in a sequence to allow the teams in each work station on the assembly line to fit the required options (e.g. radio, sunroof) on the cars within the capacity of that work station. In unsolvable problems, one would like to minimize the penalties associated to the violation of the capacity constraints. Previous attempts to tackle the problem have either been unsuccessful or restricted to solvable CarSPs only. In this paper, we report on promising results in applying a generic *genetic algorithm*, which we call GAcSP, to tackle both solvable and unsolvable CarSPs.

1 Introduction

1.1 Overview

Constraint satisfaction is a general problem which is found in many areas. A *constraint satisfaction problem* (CSP) is a problem in which one would like to assign a value to a set of variables, satisfying a set of constraints (the CSP will be defined more formally later). The generality and importance of constraint satisfaction has led to active research in this field in recent years and the development of commercial constraint problem solvers, such as CHIP and ILOG Solver (Cras, 1993). One of the areas in which success has been reported is scheduling. In industrial scheduling, resources are typically

scarce, and therefore, many CSPs have no solution. A *Partial constraint satisfaction problem* (PCSP) is a problem in which constraints may be violated at certain pre-defined costs (Freuder & Wallace, 1992) (Wallace & Freuder, 1993).

Typical algorithms for solving PCSPs are variants of branch-and-bound, which application is limited by the combinatorial explosion problem because of the NP-completeness nature of PCSPs. In Freuder *et. al*. 1995), Tsang argued for the role of genetic algorithms in partial constraint satisfaction. In this paper, we present a generic *genetic algorithm* (GA) strategy which we call GAcSP, which is designed to tackle PCSPs. GAcSP is a combination of a GA with repair and hill-climbing. GAcSP has been domstrated to be successful in another PCSP, namely, the processor configuration problem (Warwick & Tsang, 1993). In this paper, we describe its application to the *car sequencing problem* (CarSP), which is a version of the *job-shop scheduling problem*. The CarSP is known to be NP-complete. It was seen as a challenge to *artificial intelligence* (AI) (Parrello, 1988).

PCSP and GA are summarized in Sections 1.2 and 1.3. The objective of this research is described in Section 1.4. The CarSP is described in detail and formalized as a PCSP in Section 2. GAcSP is described in Section 3. In Section 4 we present empirical results on testing GAcSP on both solvable and unsolvable CarSPs. We compare the performance of GAcSP with other heuristic techniques which are applicable to solvable as well as unsolvable PCSPs. Section 5 concludes the paper.

1.2 Partial Constraint Satisfaction Problems

The constraint satisfaction problem (CSP) is an important class of problems in AI and computer science (Tsang, 1993; Freuder & Mackworth, 1994). Instances of CSPs include scheduling, scene labeling, graph isomorphism, boolean satisfiability and graph coloring. The CSP comprises a finite set of variables, each of which has a finite domain, and a finite set of constraints. A solution tuple is an assignment of a value to each variable (from their respective domains) satisfying the constraints. Following (Tsang 1993), we formally define a CSP as follows:

Definition 1 (CSP): A constraint satisfaction problem (CSP) is a triple:

$$(Z, D, C),$$

where Z = *set of variables* $\{x_1, x_2, ..., x_N\}$*;*

D = *a function which maps every variable* x_i *to a discrete domain* D_i*;*

and C = *a set of constraints on an arbitrary subset of variables in* Z*, restricting the values that they can take simultaneously.*

In other words, each discrete variable x_i has a domain $D_i = \{v_{i1}, v_{i2}, ..., v_{ik}\}$ where cardinality $k = |D_i|$. A *label* is an assignment of a value to a variable. A *compound label* is a set of labels for a set of variables.

The PCSP is an optimization problem, where an objective function *g* is defined which maps every compound label to a numeric value. The task is to find a compound label with the optimal (which can be maximal or minimal, depending on the problem specification) value. In other words, PCSPs are CSPs where there may not be a solution which satisfies all the constraints, in which case the requirement is to find "the best" solution tuple which minimizes or maximizes the objective function. Later we shall show that the CarSP is an instance of a PCSP.

1.3 Genetic Algorithms

GAs are stochastic search techniques which explore combinatorial search spaces using simulated evolution (Holland, 1975). Exploration is achieved through the recombination of data structures (which represent candidate solutions) which are given a *fitness* value according to a domain specific objective function. Selection of data structures from a population based upon the relative fitness of the data structures exploits those that are more successful in minimizing or maximizing the objective function. GAs can converge to near optimal solutions, but generally lack a local improvement ability. In this research, we test a strategy named GAcSP which combines the GA robust search technique with a local improvement ability. We argue that such a combination provides an effective approach for tackling PCSPs.

1.4 Motivation and Objective

In some scheduling problems, such as resource allocation, we would like to optimize certain cost or utility. These problems are CSPs with the additional requirement of optimizing a domain specific objective function. PCSPs are difficult because they effectively require all solutions to be found and compared in order to find an optimal solution. PCSPs with tight constraints can be tackled by complete methods (e.g. branch-and-bound) where efficient heuristics can reduce the size of the space to be searched. On the other hand, PCSPs with loose constraints have a much larger proportion of the space to be searched and are therefore potentially more difficult. Methods for tackling PCSPs are faced with the difficulty of having to compare all solutions to find the one which violates the constraints of the least cost (should constraints need to be violated).

When complete search methods cannot be expected to obtain solutions to PCSPs within a reasonable time period because of the combinatorial explosion problem or lack of efficient heuristics, stochastic methods can be used. Stochastic methods such as GA, *Heuristic Repair* (HR) (Minton, et. al. 1990), GENET (Davenport *et. al.* 1994) or GSAT (Selman *et. al.* 1992, 1993) are incomplete search techniques which sacrifice completeness and settle for near optimality to be achieved in an acceptable period of time. GAs have been demonstrated successful on combinatorial optimization problems (such as TSP and QAP), and have shown promise when applied to constraint optimization problems (Tsang & Warwick, 1990) (Michalewicz *et. al.* 1989, 1991). Motivated by the generality and importance of PCSPs, the objective of this research is to develop a generic GA-based tool for tackling them efficiently.

2 The Car Sequencing Problem (CarSP)

2.1 Definitions

In a CarSP, one is given a set of pre-defined car types, each of which requires a different set of options (e.g. car radio, seat covers etc.) to fitted by specialized teams in workstations on an assembly line. The task is to sequence a specified number of cars for each car type so that workstation teams can fit the required options whilst the scheduled cars pass through the workstations. For k car types, there are

pr[1], ..., pr[*k*] *production requirements* of the CarSP. We can calculate the total number *N* of cars to be sequenced using Equation 1:

$$N = \sum_{j=1}^{k} pr[j] \quad (1)$$

The complexity of a CarSP is thus N^k. We define a set *O* of option requirements for an *n* option CarSP with *k* car types as: *O*[*m*,*j*] for all *m* = 1, 2, ..., *n* and *j* = 1, 2, ..., *k*. O[*m*,*j*] = 1 if option *m* is required by car type *j*; 0 otherwise.

For each option *m* there is a specialist workstation on the assembly line with a team or teams of workers which can fit p_m options in the time it takes for q_m cars to pass through the workstation. This represents the *capacity constraint* $p_m : q_m$ on that option. We can calculate the number of option *m* required in a schedule:

$$O_{num}(m) = \sum_{j=1}^{k} (pr[j] \times O[m, j]) \quad (2)$$

Also, the maximum number of option *m* allowed in a schedule by the capacity constraint p_m:q_m (for simplicity, we assume that *N* is devisible by q_m) is:

$$O_{max}(m) = \frac{p_m}{q_m} \times N \quad (3)$$

The level of resources utilization in the workstation for option *m* can be measured by the *utility ratio for m* as:

$$u_m = \frac{O_{num}(m)}{O_{max}(m)} \quad (4)$$

A necessary but not a sufficient condition for a CarSP to be solvable is that all capacity constraints p_m:q_m must be satisfiable, that is $\forall$m:($u_m \leq 1$).

The overall level of resources utilization in a CarSP can be characterized by the *average utility*:

$$\mu = \frac{\sum_{m=1}^{n} u_m}{n} \quad (5)$$

We can demonstrate these ideas with a simple example CarSP presented in Table 1.

Table 1: Example of a solvable CarSP

option	car type 1	2	3	4	capacity constraint *p:q*	O_{max}	O_{num}	u_m
1 car radio	1	0	0	1	1:2 = 0.50	6	5	0.83
2 furry dice	0	1	0	1	2:3 = 0.66	8	6	0.75
3 power steering	0	0	1	0	1:3 = 0.33	4	4	1.00
production requirement (*pr*[*i*]):	2	3	4	3			μ =	0.86

In this problem, 12 cars of four types need to be produced, and three options are available. Car radio (option 1) needs to be fitted in cars of types 1 and 4. Two cars of type 1 and three cars of type 4 must be produced, and therefore $O_{num}(1) = 2 + 3 = 5$. The capacity constraint for car radio is 1:2, or 50%. A total of 12 cars need to be scheduled. Therefore, $O_{max}(1) = 50\% \times 12 = 6$. The following example schedule *S* satisfies the capacity constraints in Table 1:

position *I* in *S*:	12	11	10	9	8	7	6	5	4	3	2	1	
car type (1 to *m*):	4	3	4	2	1	3	2	1	3	2	4	3	→ assembly line

In this schedule, position 1 is assigned to car type 3, position 2 to car type 4, etc. This schedule satisfies the capacity constraint of, say, car radio (which is named option 1). This is because p_1:q_1 is 1:2, and no two consecutive positions are assigned to car types 1 and 4 (which, according to table 1, are the only car types which require radios to be fitted).

2.2 The Penalty Function

There are CarSPs which are not solvable because the capacity constraints cannot be satisfied (i.e. $u_m > 1$). In these problems, *penalty functions* are used to minimize the capacity constraint violation and encourage spacing between options (Parrello *et. al*. 1986). Adding option 3 to type 1 cars will make the previous example CarSP unsolvable as in Table 2 (differences from Table 1 are in bold).

Table 3: Summary GAcSP, HR and TABU Experiment 4.2 Results

option	car type 1	2	3	4	capacity constraint p:q	O_{max}	O_{num}	u_m
1 car radio	1	0	0	1	1:2 = 0.50	6	5	0.83
2 furry dice	0	1	0	1	2:3 = 0.66	8	6	0.75
3 power steering	**1**	0	1	0	1:3 = 0.33	4	6	**1.50**
production requirement (*pr*[*i*]):	2	3	4	3			μ =	0.86

In the Table 2 CarSP there are $O_{num}(3) = 6$ cars requiring option 3, yet the maximum allowed is $O_{max}(3) = 4$, therefore the utility ratio $u_3 = 6/4 > 1$ and the CarSP is unsolvable.

Using option 3 we can demonstrate that we cannot satisfy the capacity constraint 1:3 in Table 2:

position *i*:	12	11	10	9	8	7	6	5	4	3	2	1	
option 3 position:			✓			✓			✓			✓	→ assembly line

In this schedule, we have positioned the four type 3 cars. We cannot position any of the two type 1 cars without violating the capacity constraint. Type 1 cars need to be positioned in such a way as to minimize the capacity constraint violation according to a penalty function. For each car requiring option *m*, there is a sub-sequence of (q_m −1) cars which follow it in a schedule, defined as an *interval of relevance* (Parrello, 1988). The penalty function assigns a penalty value to the car requiring option *m*, depending upon the number of cars in the interval of relevance requiring *m* which exceed the workstation capacity p_m . A set P of penalty values is defined for each option *m* and *o* cars requiring this option in the interval of relevance:

$$P[m,o] \text{ for } m = 1, 2, ..., n;\ o = 1, 2, ..., (q_m - 1).$$

Naturally, if the capacity constraint of option *m* is p_m:q_m , then $P[m,o] = 0$ whenever $o < p_m$. The penalty cost of option *m* for car *i* in a schedule *S* is calculated as:

$$cost(S,i,m) = P[m,(O[m,S_i] \times \sum_{j=i+1}^{i+(q_m-1)\le N} O[m,S_j])] \quad (6)$$

where S_i is the type of the *i*th car in S. $O[m,S_i]=1$ if car *i* requires option *m*; 0 otherwise.

It follows, that if for all $m = 1, 2, .., n$ and $o = p_m+1, ..., (q_m - 1)$ $P[m,o]=1$, then Equation 6 calculates the total number of options violated by a single car.

Further, it may be possible to improve the car spacing arrangement in a schedule, which can assist the workstation teams install the options. Within the interval of relevance, Parrello (1988) defines a smaller *proximity interval* for *n* options as, I[1], I[2], ..., I[*n*] where $I[m] < q_m$. If a car has a penalty cost for an option *m* greater than zero (i.e. $cost(S, i, m) > 0$) and if there is another car which requires *m* within a proximity interval I[*m*], i.e.:

$$(\sum_{j=i+1}^{i+I[m]\le N} O[m, S_j]) \ge 1$$

then a proximity factor F[*m*] is added to the penalty cost for that car:

$$T_{cost}(S, i, m) = cost(S, i, m) + F[m]. \quad (7)$$

For example using I[3]=1 and F[3]=7, we find that by adding the proximity factor to the following example we can see how pressure is placed on cars requiring option 3 to have non-option 3 cars to come between them. Car 1 has car 2 requiring option 3 within I[3]=1 cars and car 1 has a *cost*(*S*, 1,3) = 2, therefore T_{cost} (*S*, 1, 3) = (2 + 7) = 9:

position *i*:	...	4	3	2	1	
require option 3?	...	✗	✗	✓	✓	→ assembly line

Spacing the cars requiring the same options apart prevents the additional proximity factor from increasing the cars' penalty cost. With a proximity interval of 1, swapping the car in position 2 and car position 3 as in the following results in a cost T_{cost} (*S*, 1, 3) =2 (since no car in the proximity interval requires option 3):

position *i*:	...	4	3	2	1	
require option 3?	...	✗	✓	✗	✓	→ assembly line

The cost of *N* cars in *S* with *n* options gives schedule cost:

$$S_{cost} = \sum_{i=1}^{N} \sum_{m=1}^{n} T_{cost}(S, i, m) \quad (8)$$

S_{cost} represents the sum of penalties for all cars which violate the capacity constraints and proximity intervals. A schedule can be derived from the problem in Table 1 with S_{cost} = 0.

In constraint satisfaction terms the production requirement is a *hard* constraint, and the capacity constraint is a *soft* constraint (which can be violated at a cost).

2.3 Theoretical Lower Bound

In order to test the quality of GAcSP results on unsolvable CarSPs, we have devised a method to calculate a *theoretical lower bound* for certain unsolvable CarSPs. The applicability of this lower

bound formula is limited to problems produced in the following way: solvable CarSPs (i.e. $S_{cost} = 0$) made unsolvable by a single over-utilized option. For simplicity, we make P[m,o] = 1 and F[m] = 0 for all m and o (see example in table 2).

We can calculate the required number of options in the schedule, $O_{num}(m)$, in excess of the maximum number allowed, $O_{max}(m)$, by the capacity constraint p_m:q_m as:

$$O_{exe}(m) = O_{num}(m) - O_{max}(m).$$

After all $O_{max}(m)$ options have been sequenced to satisfy p_m:q_m , the remaining $O_{exe}(m)$ options need to be placed in the remaining spaces so as to minimize the capacity violation. If we add two extra options to an interval of relevance as follows:

position i:	12	11	10	9	8	7	6	5	4	3	2	1	
option 3 required:			✓			✓			✓	✓	✓	✓	→ assembly line
violation:										↑	↑	↑	S_{cost} = 3

where p_3:q_3 = 1:3, this will result in a penalty of 3 (due to the cars in positions 1 to 3), which is also the minimum violation cost (see Warwick 1995). However if we do not group the extra options in one interval of relevance, as in the following example:

position i:	12	11	10	9	8	7	6	5	4	3	2	1	
option 3 required:			✓	✓		✓			✓	✓		✓	→ assembly line
violation:				↑		↑				↑		↑	S_{cost} = 4

then a greater cost (in this case 4, due to positions 1, 3, 7 and 9) will result. These examples show that the grouping of extra options in available spaces in the minimum number of intervals of relevance reduces the number of violations. Consequently, $p_m - q_m$ spaces in each interval of relevance can be used to accommodate extra options. We can therefore calculate the minimum number of violation as follows:

$$\frac{O_{exe}(m)}{q_m - p_m} \times q_m .$$

In addition, an extra option m placed in a space at the end of S presents a special case where only one violation occurs, for example:

position i:	12	11	10	9	8	7	6	5	4	3	2	1	
option 3 required:	✓		✓			✓			✓			✓	→ assembly line
violation:			↑										S_{cost} = 1

Allowing for this special case, the lower bound formula becomes:

$$lowerbound = \frac{O_{exe}(m)}{q_m - p_m} \times q_m - p_m \quad (9)$$

3 Outline Of GAcSP

An outline of GAcSP is given in Figure 1. GAcSP is distinguished from the standard or simple GA (Goldberg, 1989) by the integration of elitism (De Jong, 1975), adaptive template type crossover (Syswerda, 1989), repairing and hill-climbing (HC) into a single strategy. The reproduction operator encourages exploitation of population information by the use of elitism and fitness biased selection. Exploration is achieved through the *uniform adaptive crossover* (UAX), which uses parent binary templates to control offspring creation. By utilizing matching parent template values (as opposed to single template crossover points as in Schaffer and Morishima (1987)) we hope to enable high fitness constraint links between parent values to be inherited.

After crossover, the offspring is repaired and hill-climbed. The repair function and HC act as mutation operators altering individual string elements. Although the GA is a robust technique for finding near optimal solutions in combinatorial search spaces, it generally lacks a local improvement ability. We provide this local ability by combining the GA with a simple string element exchange function (HC). HC increases the potential for every offspring after crossover generation, before the string is expected to compete with its peers. The combination of a GA and HC is synergistic, exploiting the abilities of each method. The representation and the operators of GAcSP are described in the following sections.

3.1 Representation

GA operators manipulate artificial chromosomes in the form of string-like data structures. PCSPs can best be handled by real-coded (Goldberg, 1990) data structures - where string positions represent PCSP variables and string elements are values from the corresponding domains. In this section, we formally define the CarSP as a PCSP. Then we shall propose a specialized GA representation for tackling this problem. We argue that this representation is applicable to PCSPs in general.

Definition 2 (**CarSP)**:

A car sequencing problem (which we refer to as CarSP) is a partial constraint satisfaction problem (Z, D, C, g*) where:*

the variables are the positions in the sequence: Z = {1, 2, ..., N*};*

all variables have the same domain, which is the set of car types: $\forall$ x$\in$ Z*:* D_x *= {1, 2, ...,* k*};*

the set of constraints C comprises:

production requirements*: pr[*i*], where* i *= 1 to the number of car types,* k*, and*

capacity constraints: p_j*:*q_j *, where j = 1 to the number of options, n;*

the cost function g is S_{cost} *in equation (8) with P[m,o] = 1 and F[m] = 0 for all m = 1, ..., n, and o =* p_m *+ 1, ...,* (q_m *– 1).*

The GAcSP objective function *g* maps each CarSP solution tuple to a numerical value, which is often called the *fitness*. The goal of GAcSP is to find optimal or near optimal solution tuples to the CarSP which minimize the fitness.

3.2 Reproduction Operator

The reproduction operator guides the GA through the search space by selective control using a sampling bias based upon the string fitness. The first stage of the operator implements a technique called elitism which copies the strings with the best fitness (i.e. lowest costs in CarSP) into a mating population, called the *matepool*. This technique guarantees that the elite members of the population will survive into the next generation. These best fitness strings are important because they will have low cost elements or groups of elements (i.e. *building blocks*) in their strings to pass onto their offspring, which will direct the search towards optimal regions of the search space. The second stage of reproduction involves a fitness-biased selection from the population.

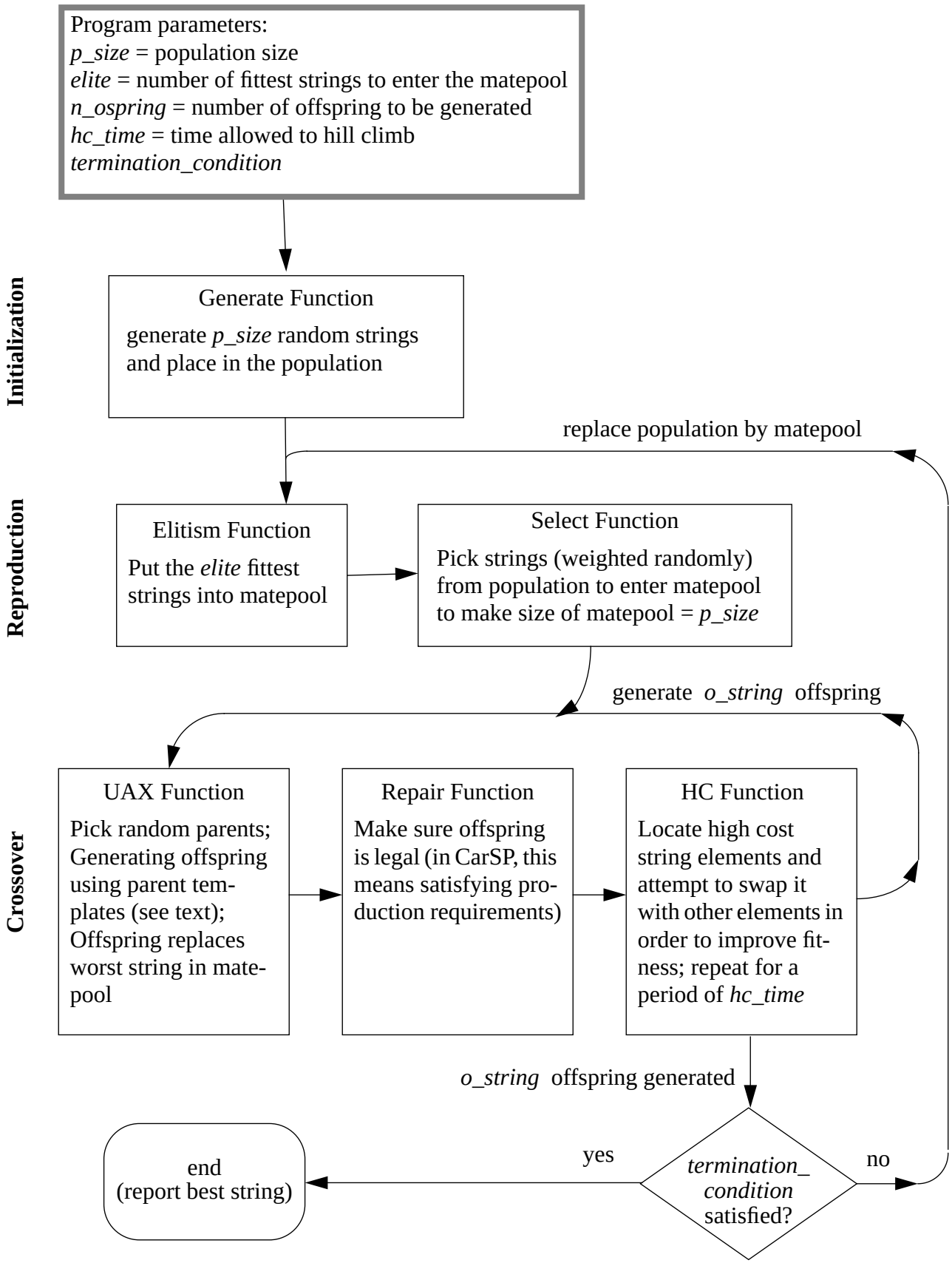


Figure 1: Overview of GAcSP Operators and Control Flow

3.3 Crossover Operator

The GA crossover operator explores the structural search space by creating offspring strings from selected parent strings. A crossover operator needs to encourage exploration, yet not destroy the important information already contained in the population. The crossover operator should allow the offspring to inherit building blocks from the parents. GAcSP uses a uniform adaptive crossover (*UAX*) (Warwick & Tsang, 1993), which has an extended string representation. It is designed to exploit PCSP constraints by enabling links between string values to be inherited. The UAX is suitable for PCSPs because in a PCSP, the variables have no inherent ordering, but the value for each variable is highly dependent on the values for a set of other variables (due to the constraints). Represented in a string, the variables are given a particular order. In a standard crossover, short schemas are less likely to be destroyed. By using the UAX, we hope that break off points which reflect the dependency relations of the variables can be discovered and passed on to future generations.

The extra binary string acts as a template to control the creation of the offspring string during the crossover process. Successful strings will have the opportunity to become parents and pass their crossover points on to offspring. The first stage of the crossover operator weighted randomly selects two parents from the matepool. Parent strings are cut after each matching crossover point (to be determined by the parent templates) and alternating sections are used to create an offspring. At start, an offspring inherits the values from a randomly selected parent. This continues until the templates of both parents share the same value. When this happens, the offspring inherits values from the other parent, until the next common value is shared by the two parent templates. The following example should make this process clear:

	string position:	1	2	3	4	5	6	7	8	9	10
parent 1	string solution:	1	2	3	4	2	5	4	1	3	5
	template:	0	1	0	**0**	1	**1**	0	**0**	1	0
parent 2	string solution	2	1	3	2	4	1	5	3	4	5
	template	1	0	1	**0**	0	**1**	1	**0**	0	1

The offspring generated from parents 1 and 2 is:

	from parent:	1	1	1	2	2	1	1	2	2	2
offspring	string solution:	1	2	3	4	2	5	4	1	3	5
	template:	0	1	0	0	1	1	0	0	1	0

This offspring first inherited values from parent 1. At position 4, both parent templates share the value 0. Therefore, the offspring started to inherit values from parent 2. Two more switches were made at positions 6 and 8. The offspring replaces the lowest fitness member of the population.

3.4 Repair and Hill Climbing Operators

One effect of the crossover operator upon the representation is that offspring created will not always satisfy the CarSP constraints (i.e. production requirements). We ensure each offspring satisfies the production requirements by using a greedy repair function. This function is necessarily application dependent.

For the CarSP, we defined a greedy repair function which works in the following way: it first searches in the string for values which are over-represented (> pr[*j*]) and values which are under-represented (< pr[*j*]). Then an arbitrary set of string positions which take the over-represented values are selected, and their values replaced by under-represented values. This ensures that the string represents a schedule which satisfies the production requirements, which is a hard constraint.

After repair, each offspring is hill-climbed by a string element swap function for a pre-set time period. In each iteration of the hill climbing, an arbitrary pair of string positions are picked. If the swapping of values between these two positions result in a fitter string, then the swap will be accepted, and hill climbing continues from the new string. The same strategy is later used with success in the connectionist approach Genet (Davenport & Tsang 1995).

4 Empirical Results

4.1 Experiments overview

In our experiments we are concerned with GAcSP's ability to cope with CarSPs with both loose and tight constraints (Sections 4.2 and 4.4), various sizes (Section 4.3) and over-constrained (hence unsolvable) problems (Section 4.4). All solvable CarSPs were generated by a program supplied by Kangmin Zhu which provided a solution to each problem satisfying the capacity constraints (Zhu,

1993). All CarSPs tested have 5 options with capacity constraints; 1:2, 2:3, 1:3, 2:5 and 1:5. This range of capacity constraints allows us to test GAcSP performance and directly compare our results with those of other researchers.

Recently, Chow *et. al.* (1992) applied simulated annealing to the car sequencing problem; but since it uses a different formulation of the problem their results are not directly comparable with ours. Other work which apply GAs to CSPs include Eiben *et. al.* (1994) and Filipic (1992). GSAT or its extensions (Selman *et. al.* 1992, 1993, 1994) have not been included in our tests because adapting it to the CarSP is a non-trivial task.

All algorithms were written in C and tests were run on SUN 4/110 work-stations under the UNIX 4.0 operating system. The following parameters have been used for GAcSP throughout all the tests: (a) population size is 80, which was found to be effective in an earlier work (Tsang & Warwick, 1990); (b) 10% of the fittest members (elite) of the population were copied directly into the mating pool at reproduction phase of GAcSP; (c) The number of offspring created in each cycle was arbitrarily set at four; (d) the termination conditions are 400 cycles or 10 CPU hours; and (e) a maximum of 30 CPU seconds is allowed for hill climbing for each offspring.

It should be emphasized here that algorithms comparison is difficult in general. Run time can be seriously affected by the ways that algorithms are implemented. Besides, our comparison in experiment 4.2 is limited by the capacity of the algorithms that we compare GAcSP to.

It may be worth emphasizing that tabu search is in fact a class of algorithms. The instantiation of the tabu list plays a crucial part in its effectiveness and efficiency. The instantiation that we used in the comparison below is the most successful one that was developed in (Zhu, 1993).

4.2 GAcSP, HR and Tabu Tackling CarSPs of Different Tightness

In *Experiment 4.2* GAcSP, *Heuristic Repair* (HR) (Minton *et. al.* 1990) and a version of *Tabu search* (Tabu) (Glover, 1989; Glover, 1990) were tested on solvable 100 car CarSPs with average utilities μ ranging from .45, .50, ..., .90. The HR strategy assigns a random value to each variable, and then repeat the following steps: pick a variable whose current value violates some constraints, and re-assign to it a value which minimizes the number of constraints violated (which could result in

assigning the same value to it). This process terminates when no constraint is violated or resources (e.g. a maximum number of iterations) have run out. Tabu search is a local search strategy which uses a tabu list to restrict the moves for transforming one solution (state) into another. Both of these algorithms were adapted to tackle solvable and unsolvable CarSPs. The tabu search used in our experiments is identical to HR, except that the value that has been replaced in the preceding iteration will not be used in the current iteration.

The pseudo code for HR and the version of tabu search used in our experiments are presented in Appendix A. We compare GAcSP against HR and Tabu because (a) both can be extended to tackle solvable and unsolvable CarSP's; (b) like GAcSP, they can handle optimization problems (most search techniques in constraint satisfaction were developed for satisfiability problems only); and (c) they are well known algorithms in constraint satisfaction research (which motivated this work).

The iteration limit for HR and Tabu was set to 100,000 adjustment cycles. It was found (empirically) that allowing more iterations (say 1,000,000) did not improve the quality of the best results (Zhu, 1993). For each of the 10 average utilities tested, we randomly generated 10 solvable CarSPs, and 10 runs were carried out on each problem. Therefore, there were a total of 100 runs for each utility test. (HR and Tabu results were supplied by Dr. Zhu.) The experimental results are summarized in Table 3.

Table 3: Summary GAcSP, HR and TABU Experiment 4.2 Results

avg utility μ avg car types *k*	.45 8.7	.50 12.3	.55 12.9	.60 16	.65 18.2	.70 20	.75 21.2	.80 22	.85 23.4	.90 23.3
GAcSP - number solns	100	100	99	100	99	99	92	61	21	1
GAcSP - avg violation	0.00	0.00	0.01	0.00	0.01	0.01	0.09	0.50	1.40	3.90
GAcSP - avg run-time sec	29	49	69	43	60	212	457	1122	2104	4421
HR - number solns	98	97	94	96	94	97	88	58	15	1
HR - avg violation	0.02	0.04	0.07	0.08	0.07	0.04	0.19	1.04	2.42	7.00
HR - avg run-time sec	19	26	46	40	57	44	144	451	856	975
TABU - number solns	100	100	100	100	100	100	97	21	0	0
TABU - avg violation	0.00	0.00	0.00	0.00	0.00	0.00	0.05	1.62	5.74	11.85
TABU - avg run-time sec	4	6	11	4	8	10	111	818	956	960

The following keys are used in tables 3 to 5:

avg car types *k*	The average number of car types for each average utility
number solns	Number of runs returning solutions (out of 100), i.e. where $S_{cost} = 0$
avg violation	The mean of minimum violations in the 100 runs (including solutions)
avg run-time *sec*	The mean of run-times (in CPU seconds)

For each algorithm, the number of times that it returns a solution, the number of constraints violated in the best (partial) solution during each run, as well as the run time are measured. The statistically significant difference between the number of solutions found by GAcSP and HR (see Table 4 and Figure 2) demonstrates that GAcSP out-performed HR in finding solutions to Experiment 4.2 CarSPs. GAcSP has found solutions in all runs for the .45, .50, and .60 average utility tests and found 99% for .55, .65 and .70 average utility tests. GAcSP average performance for finding solutions to .45 to .70 tests is 99.5%. Tabu has found solutions to all runs in the .45 to .70 utility tests. Across all utilities, GAcSP found solutions in 77.2% of its runs. HR and Tabu found solutions in 73.8% and 71.8%, respectively, of their runs.

Table 4: Summary of F-tests on results in Experiment 4.2

	Observed F value			Critical F values	
	number solns	avg violation	avg run-time	$\alpha = 0.01$	$\alpha = 0.05$
GAcSP Vs HR	**(36.63)**	2.73	2.99	10.6	5.12
Conclusion: there is a 99% level of confidence to reject the hypothesis that there is no difference between the performance of GAcSP and HR in finding solutions					

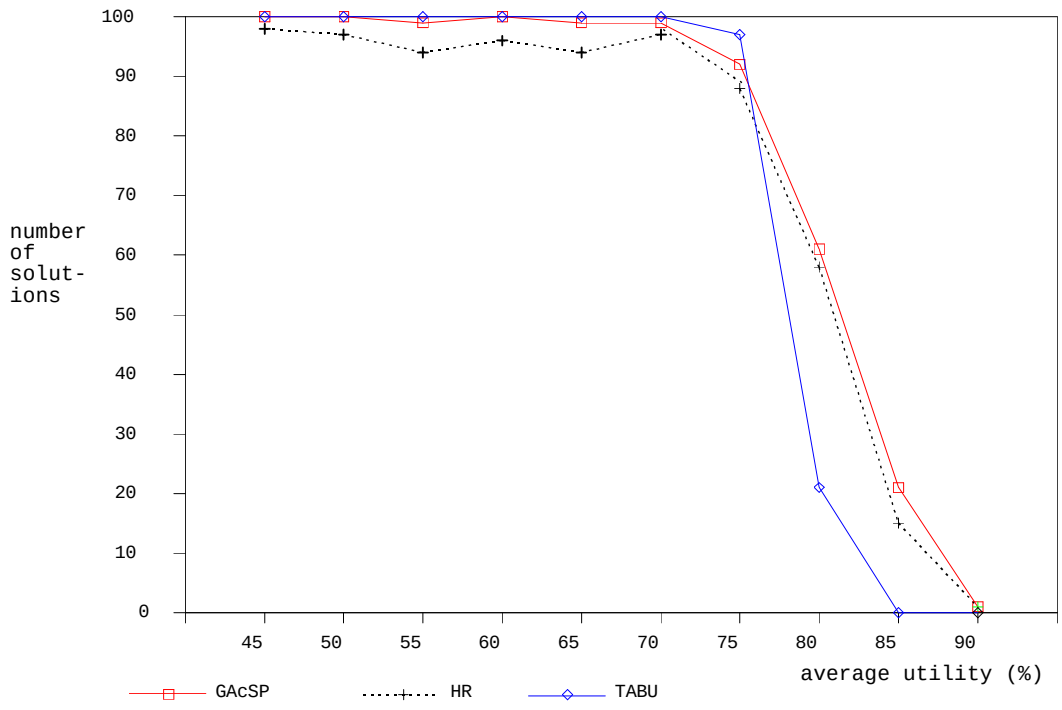


Figure 2: Experiment 4.2, GAcSP, HR and Tabu on 100 cars CarSPs

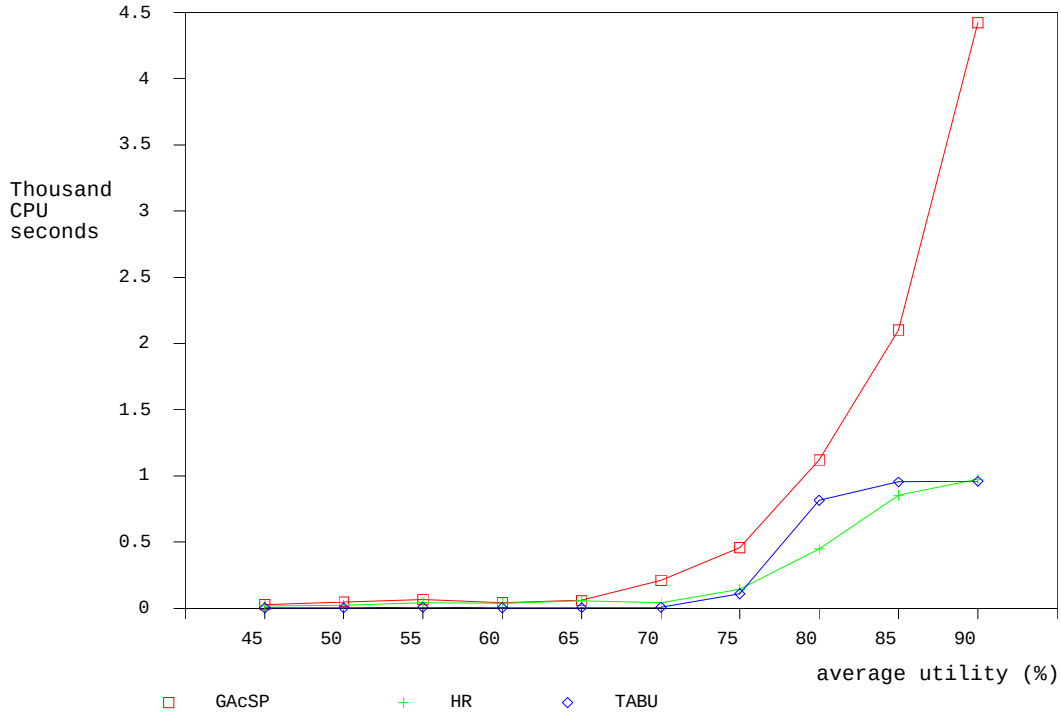


Figure 3: Average CPU time for CarSPs in Experiment 4.2

GAcSP does better in finding solutions and in minimizing the number of violations than HR and Tabu when the algorithms are put under pressure by the increasing complexity of CarSPs. Both GAcSP and Tabu are able to find solutions in nearly all .45 to .75 average utility runs within reasonably quick run-times (see Figure 3). However, it is only at the .80 to .90 utility results that we can clearly distinguish between the behaviors of the algorithms tested. At .80 all algorithms suffer a severe reduction in solution finding ability, with Tabu failing to find any solutions at .85 and .90. Both HR and GAcSP find more solutions than Tabu from .80 to .90, with GAcSP finding more solutions than HR.

The dramatic reduction in performance of the algorithms on the .80 average utility test is due to the interactions between the options, which make CarSPs more difficult to solve. This option interaction is due to a combination of the number of options in the car types and the capacity constraints. GAcSP, HR and Tabu depend upon local information to explore the search space. With increasing option interactions local information is less effective in guiding the search towards solutions and

consequently the starting points used become more important. Local information becomes less helpful when a car in a schedule can violate more than one option, which creates two difficulties for local search techniques: (1) there are fewer alternative positions to move cars to in order to reduce their option violations; and (2) there are more cars with option violations.

Both HR and Tabu will suffer from their dependence on the quality of good starting points. Tabu is expected to perform better than HR because it can escape from local minima, due to a limited memory for previous choices. With GAcSP, increasing CarSP utility provides reduced feedback about the fitness space used to guide both the GA and HC components. But information from a population of search points reduces the chance of GAcSP being trapped in local minima. Although HC will contribute less directly to the search process it will still assist in the development of good building blocks. In which case, the work of the GA component is increased. This balance of work shared between GA and HC is an important feature of GAcSP -- it improves its robustness. Furthermore, performance of GAcSP could be improved by controlling this balance by fine tuning GAcSP parameters. On the other hand, HR and Tabu mainly depend upon local information, and therefore, may be harder to improve their performance.

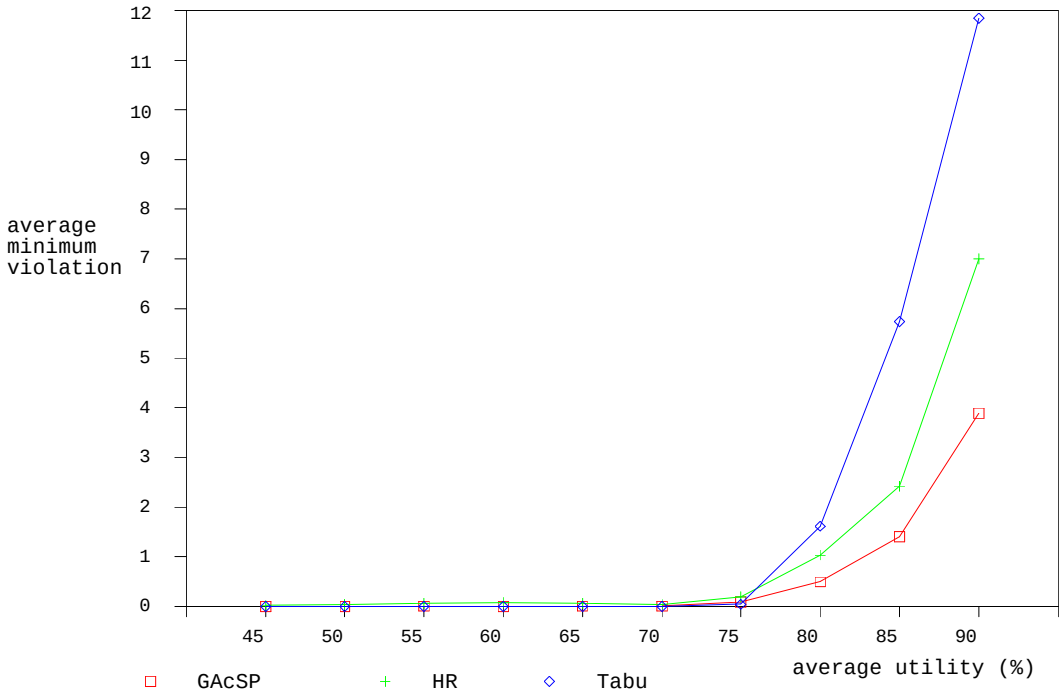


Figure 4: Average minimum violations in Experiment 4.2 results

In the .45 to .75 utility tests the average minimum violation results are dominated by the number of solutions found by the algorithms. But as the algorithms find less solutions in the .80 to .90 tests, the significance of average minimal violation as a measure of performance is increased. Although the number of solutions returned by GAcSP, HR and Tabu for increasing utility tests above .75 decline significantly, reduction in the average minimum violation is not as severe. Figure 4 shows that the rate of increase in GAcSP's average minimum violation results for .80 to .90 is not as great as those in HR and Tabu. GAcSP can remain consistently near optimal even at .90. (In Experiment 4.4 below, we demonstrate that GAcSP can retain this near optimal performance on $\mu > .90$.) The mean of average minimal violation of GAcSP, HR and Tabu on Experiment 4.2 CarSPs are 0.592, 1.097 and 1.926, respectively.

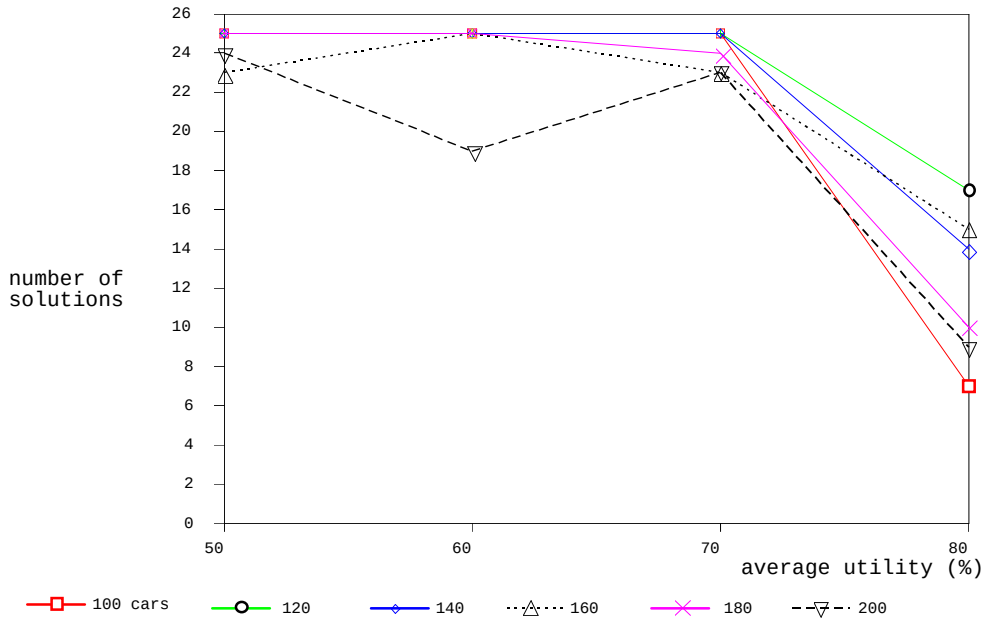
4.3 GAcSP Tackling CarSPs of Different Sizes

In Experiment 4.3 GAcSP was tested on solvable CarSPs with 100, 120, ..., 200 cars, and utilities .50, .60, .70, and .80. There were 5 randomly generated CarSPs for each utility and 5 runs carried out on each problem. Results of Experiment 4.3 are summarized in Table 5.

Table 5: GAcSP performance on CarSPs of different sizes

	.05 avarage utility						.06 avarage utility					
number cars *N*	100	120	140	160	180	200	100	120	140	160	180	200
avg car types *k*	12.6	11.2	12.2	11.4	17.8	15.2	16.4	17.4	18.2	19.6	22.6	22.2
number solns	25	25	25	23	25	24	25	25	25	25	25	19
avg violation	0	0	0	.08	0	.04	0	0	0	0	0	0.4
avg run-time *sec*	20	92	218	1228	96	421	117	214	331	916	159	3070
	.07 avarage utility						**.08 avarage utility**					
number cars *N*	100	120	140	160	180	200	100	120	140	160	180	200
avg car types *k*	20.6	21.4	22.8	21.6	25	25	22.6	22.4	24.2	24.2	25.8	262
number solns	25	25	25	23	24	23	7	17	14	15	10	9
avg violation	0	0	0	.08	0.04	.12	.92	.32	.92	1.36	.88	1.2
avg run-time *sec*	339	369	699	1704	539	4177	2033	3422	5261	7079	5969	7289

Note that a different set of 100-cars problem were generated, hence discrepancy between the results in this table and those in table 3.



(number of cars varied from 100 to 200)

Figure 5: Experiment 4.3, GAcSP on CarSPs of different sizes

The ability of GAcSP to return solutions decreases as the utility is increased, supporting our observations from Experiment 4.2 (see Figure 5). Although there is a slight reduction in the number of solutions with the increase in the number of cars, generally the ability of GAcSP is consistent. The

loss in performance is not significant (see Table 6), yet the increase in size of the search space for these CarSPs is significant (see Table 7). Table 6 shows that there is no correlation between the number of cars and the run time. This shows that the combinatorial explosion problem can be contained by GAcSP.

Table 6: Summary Experiment 4.3 result F-test statistics - significance in parentheses

	Observed F-value				Criterion F-value	
hypothesis:	number solns	avg violation	avg cost	avg run-time	$\alpha = 0.01$	$\alpha = 0.05$
Number of cars: *N*	1.87	**(5.72)**	2.21	2.79	4.56	2.90
One significant statistical conclusion from Table 6 is: there is a 99% level of confidence to reject the hypothesis that there is no correlation between the number of cars and the average violation achieved by GacSP for Experiment 4.3 CarSPs.						

Table 7: Search space sizes for Experiment 4.3 CarSPs

N^k	100^k	120^k	140^k	160^k	180^k	200^k
avg utility û = .50	1.6E+25	1.9E+25	1.5E+26	1.3E+25	1.4E+40	9.5E+34
avg utility û = .60	6.3E+32	1.5E+36	1.1E+39	1.6E+43	9.3E+50	1.2E+51
avg utility û = .70	1.6E+41	3.1E+44	8.5E+48	4.1E+47	2.4E+56	3.4E+57
avg utility û = .80	1.6E+45	3.7E+46	8.6E+51	2.2E+53	1.5E+58	1.9E+60

Since the HC time limit is held constant for all CarSPs tested, the extra work undertaken by GAcSP must be due to the GA component. This work sharing GAcSP behavior is an important design feature and suggests that the time allowed to HC depends more on problem characteristics of the number of car type options and production requirements (as we have seen with Experiment 4.2 tests) than problem size. This emphasizes the fact that the GA component of GAcSP ensures robustness whilst the HC component adds a specialist ability.

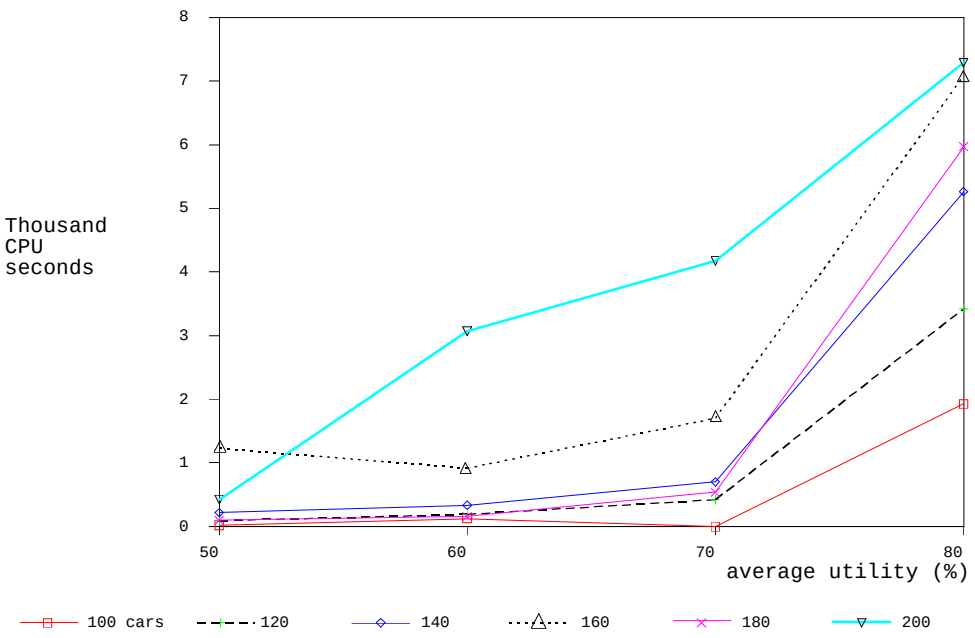


Figure 6: Average run-time for GAcSP in Experiment 4.3

In general, the ability of GAcSP in finding solutions is not necessarily restricted by search space size. However, an important effect of increasing the CarSP size is to increase the computational workload of the GA, which can slow GAcSP down. This increase in the case of GAcSP is due mainly to the CPU requirements of the evaluation function and crossover mechanism. The average CPU run-time in Figure 6 shows this increase for all run-time averages shown in Table 5 with the exception of the .50 180 car CarSP. In this case, all the test runs resulted in solutions, enabling the GAcSP to terminate before complete convergence.

We can make a limited comparison with the results from Experiment 4.2 and 4.3, with those reported by Parrello *et. al.* (1986, 1988): using an Automated Reasoning Program (ITP) and OPS5 to sequence 5 cars with 5 options took 35 minutes and 15 minutes, respectively. Dincbas *et. al.* (1988) tackled solvable CarSPs with CHIP, a *constraint logic programming* system. They reported that CHIP could sequence 100 car schedules with an average utilization of .80 in under 60 seconds and 200 cars between 336 and 345 second, but only one problem was used for each schedule. Besides, they have only tackled solvable CarSPs.

4.4 GAcSP Tackling Unsolvable CarSPs

In Experiment 4.4, GAcSP was tested on unsolvable CarSPs. Each unsolvable CarSP was generated by making a single option over-utilized (as described above). This allows us to calculate the lower bounds of the optimal costs (using equation 9).

We carried out tests on 4 groups of problems: unsolvable CarSPs were generated from solvable CarSPs with average utilities of .50, .60, .70, and .80. A total of 5 unsolvable CarSPs were generated for each group in the following way. From a CarSP in each of these groups, we produced 5 new unsolvable CarSPs by over-utilizing each of the 5 options. For each option m, where $m = 1, 2, ..., 5$, the solvable CarSP has option m added to randomly selected car types until $u_m > 1$. Five runs were made for each unsolvable CarSP. Therefore, there are a total of 25 runs for each group.

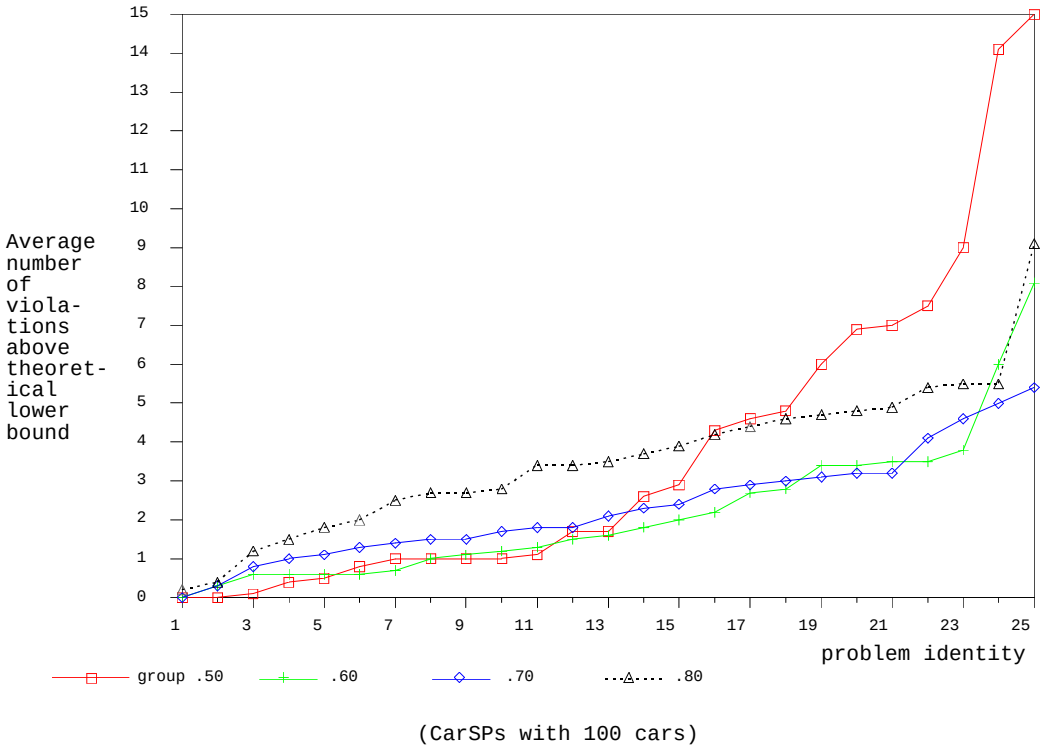
By over-utilizing a single option in creating each unsolvable CarSP we have increased the average utility significantly. For example, a number of the new average utilities are greater than .90 and in one particular case 1.024. (On average, group .50 average utilities increased by 40%; .60 by 26%; .70 by 17%; and .80 by 10%.) Yet in general, the minimum and average violation solutions are close to the theoretical lower bound (see Figure 7). We can assume that the optimal minimal violation solutions for each group of tests is within the range of these two values. GAcSP results from Experiment 4.4 have been summarized in Table 8 and Figure 7.

Table 8: Frequency, number of violations above theoretical lower bound

(25 runs were made in teach group; percentages in bracket)										
number violations	0	1	2	3	5	6	7	8	13	15
group .50 (%)	6 (24)	8 (32)	1 (4)	3 (12)		4 (16)		1 (4)	1 (4)	1 (4)
group .60 (%)	8 (32)	10(40)	5 (20)		1 (4)		1 (4)			
group .70 (%)	7 (28)	9 (36)	5 (20)	4 (16)						
group .80 (%)	4 (16)	7 (28)	5 (20)	7 (28)	1 (4)		1 (4)			

Table 8 summarizes the following results for each group of problems: (1) the number of solutions obtained that are n constraint violations above the lower bound ($n = 0, ..., 15$); and (2) the percentage of solutions in each category. The same GAcSP performance statistics were summarized for

Experiment 4.4 as for Experiment 4.3. The theoretical lower bounds were calculated using Equation 9. We can see from Table 8 that an average 25% of the theoretical optimal solutions (solutions whose costs are at the theoretical lower bound) are found, with a further average of 64% within 3 violations (see Figure 8).



(CarSPs with 100 cars)

Figure 7: Average number of violations for Experiment 4.4 CarSPs

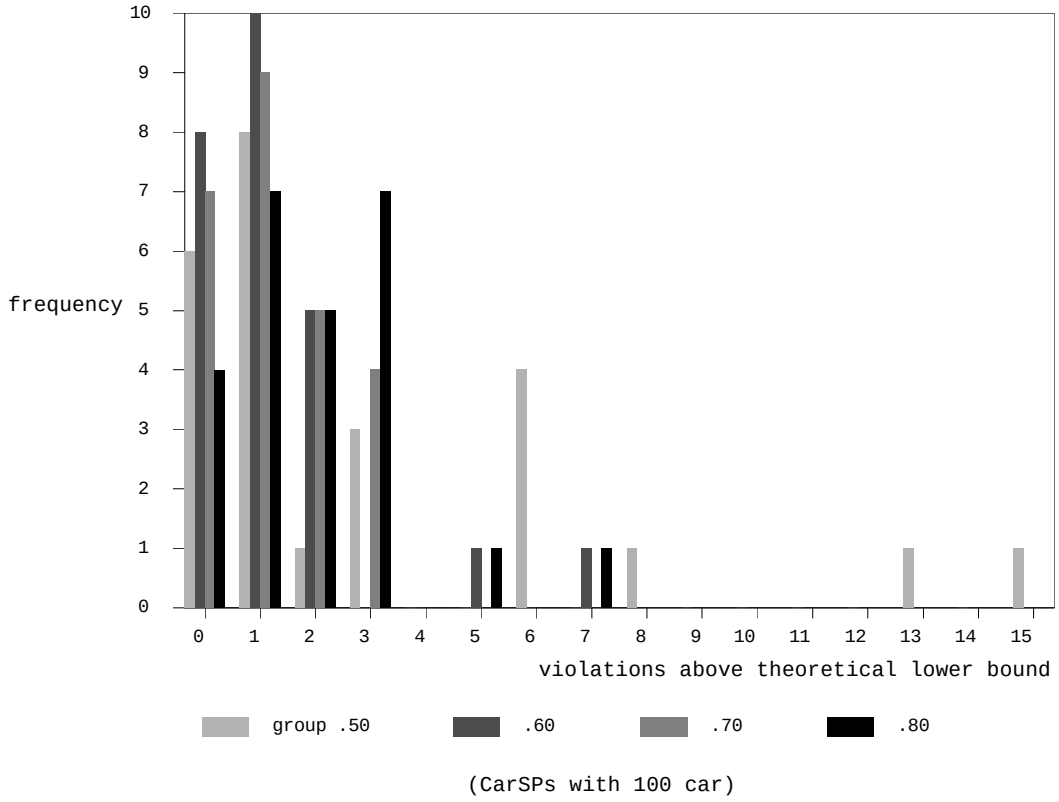
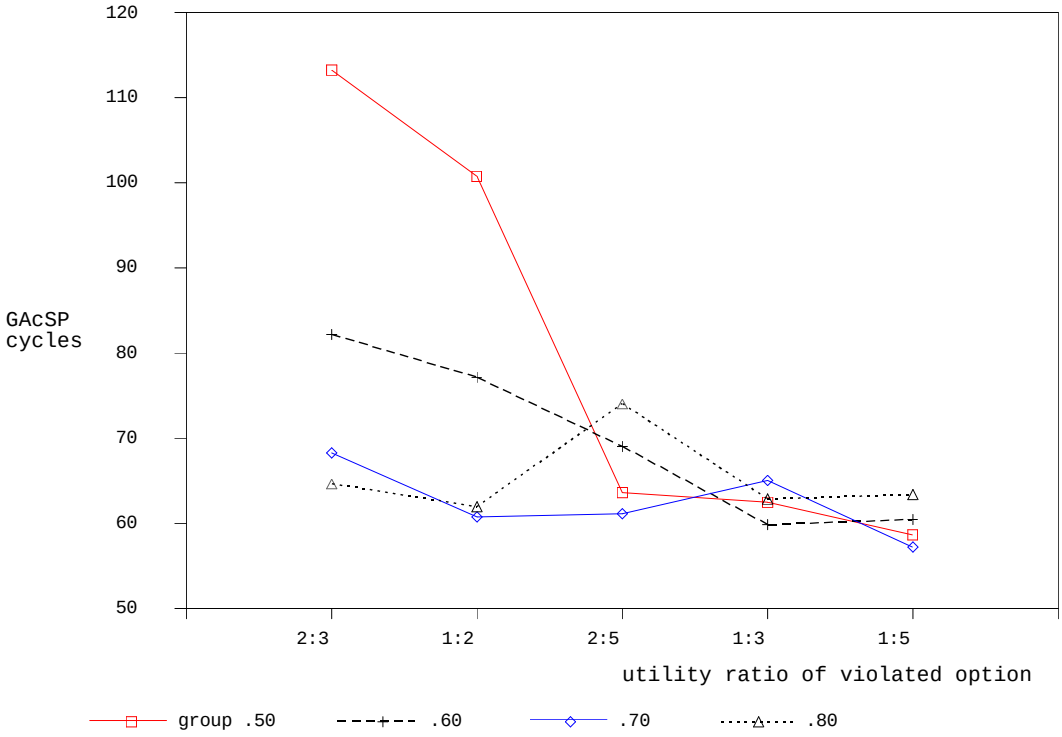


Figure 8: Summary of number of violations over lower bound

GAcSP can exploit tight utility ratio constraints to sustain the search in tackling unsolvable CarSPs, through the action of the crossover operator. In Figure 9 we present the mean number of cycles (y-axis) to convergence (to a state in which all strings have the same fitness) for each utility ratio run (a few runs were terminated at the maximum 400 cycles). The x-axis in Figure 9 represents decreasing capacity constraint tightness, measured as the number of non-option spaces allowed in a schedule by the capacity constraint:

$$\textit{capacity constraint tightness} \;=\; 1 - \frac{p_m}{q_m} \tag{10}$$

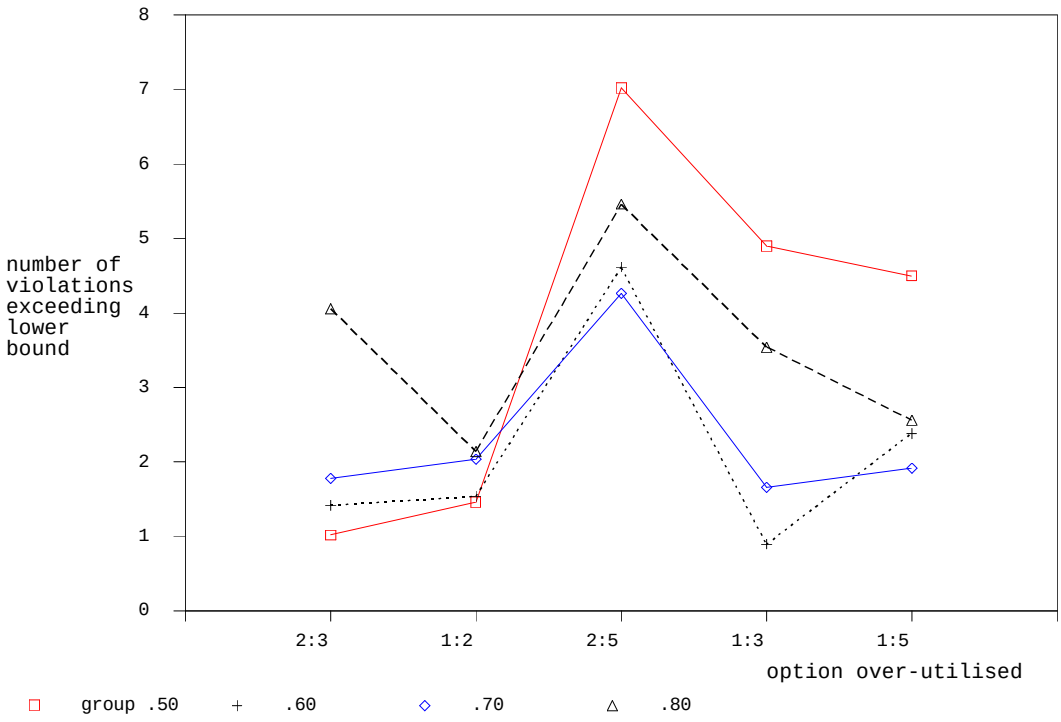
where p_m:q_m is the capacity constraint for option m.



(CarSPs with 100 cars)

Figure 9: Number of cycles by GAcSP in Experiment 4.4

In general, the number of cycles for each test utility decreases as the capacity constraint tightness decreases, demonstrating a positive correlation between utility ratio tightness and GAcSP cycles. The curves for groups .50, .60, and .70 demonstrate this correlation, but not so strongly with .80 (because the average utility tightness for .80 unsolvable CarSPs has an effect on the results). The average utility tightness reflects soft constraint interaction and influences GAcSP through the objective function. In the less tight utility ratio tests, GAcSP was unable to sustain the search as long. However, if we consider Figure 10 which summarizes the closeness to the lower bound that was achieved by each over-utilized category of problems, we find that the quality of results between tight and loose utility ratio tests is slightly improved. GAcSP was able to exploit the hard position dependent constraints in sustaining the search with increasing utility ratio tightness.



(CarSPs with 100 cars)

Figure 10: Experiment 4.4 results compared to lower bound

In order to sustain the search, the crossover mechanism must use knowledge of constraints in a purposeful way. The crossover operator can use position dependency due to tight constraints to try and form good building blocks. GAcSP can creep towards the optimal solutions by ensuring that tightly positioned options are recorded on the binary templates. With loose constraints (e.g. 2:5), GAcSP is required to select from a combination of alternative car positions. The alternative combinations increase the work required by GAcSP in finding a minimal violation. Furthermore, there may only be one combination of car positions for an option which will achieve a lower bound violation. Therefore, options which have lower capacity constraints allow more alternative arrangements in placing options in a schedule which satisfy the capacity constraint. The HC component can fine tune near optimal solutions to minimize the capacity violation in tightly constrained CarSPs. However, achieving an optimal sequence is more difficult and beyond the localized ability of the HC. Only UAX has the necessary ability to simultaneously sequence a number of cars to achieve this. Although alternatives

require more work from GAcSP to achieve the minimum lower bound, near optimal results could be found.

Run-times shown in Figure 9 are longer for Experiment 4.4 tests of the same size and average utility than those in Experiment 4.2 (from which they were derived) due to the fact that runs were terminated only after complete convergence. The intention was to ensure that the theoretical lower bound could not be improved upon, and to demonstrate typical run-times for unsolvable problems. The price to pay for tackling unsolvable CarSPs is increased computation, resulting in longer run times in comparison with the run times for solvable problems and times achieved by Dincbas *et. al.* (1988). Compromises can be made should one be prepared to sacrifice optimality for speed in unsolvable CarSPs.

5 Conclusion

Partial constraint satisfaction is a general problem. In this paper, we have presented a generic GA named GAcSP for tackling partial constraint satisfaction problems (PCSPs). We have demonstrated its effectiveness in a case study using the car sequencing problem (CarSP). The "engine" of GAcSP is a crossover operator (UAX) which remembers valuable crossover points in order to help retaining useful building blocks which may be separated in the string representation. The UAX attempts to exploit PCSP constraints by using an extended binary string representation, which encodes information about "preferred" cut off points.

The CarSP results show that GAcSP is not restricted to tackling solvable problems only, can be effective in both loosely and tightly constrained problems, is a robust search technique and is not deterred by the problem size (demonstrated in 100 - 200 car CarSPs). GAcSP out-performed both HR and Tabu, techniques which are applicable to both solvable and unsolvable CarSPs.

Through the action of the crossover operator, GAcSP can exploit the constraints to improve on solution quality. The GA component ensures robustness whilst the HC component adds a specialist ability. The balance of work between the GA and HC components can be controlled according to the scale of the problem. As larger problems are tackled, the GA component can undertake more responsibility for the search. With larger search spaces the GA component of GAcSP offers more guidance to locate the areas of hills for the hill-climber to exploit. Unlike other stochastic

optimization techniques for PCSPs, GAcSP is a robust exploration strategy which does not easily get trapped in local minima. Therefore, GAcSP could provide a useful and practical tool for tackling a class of combinatorial problems where current solving techniques are limited or infeasible.

Apart from the CarSP, GAcSP has been tested on the processor configuration problem (PCP) (Warwick & Tsang, 1993). Promising results in these tests support to our claim that GAcSP is a generic PCSP solver, which can achieve optimal or near optimal solutions to both solvable and unsolvable classes of PCSPs.

Acknowledgement

Warwick's Ph.D. research was supported by an SERC grant. The authors are grateful to Dr. Zhu for providing us with the HC and Tabu results, and the constructive and detailed comments by De Jong and the anonymous referees.

Appendix A -- Pseudo Codes

Pseudo codes for GAcSP:

```
PROCEDURE PCSP(P);
/* setup GAcSP parameters and then call GAcSP to solve the PCSP */
BEGIN
        p_size ← population size, e.g. 80;
        n_ospring ← number of offspring created each GAcSP cycle, e.g. 4;
        elite ← number of elite population members to select, e.g. 8;
        hc_time ← maximum time in CPU seconds to hill-climb offspring;
            /* when hc_time = 0, HC is switched off ), e.g. hc_time can be set to 30 */
        terminator ← any termination condition, e.g. maximum cycles 400;
        GAcSP((P), p_size,  n_ospring, elite, hc_time, terminator);
END /* PCSP */

PROCEDURE GAcSP((Z, D, C, g), p_size,  n_ospring, elite, hc_time, terminator);
/* main procedure to solve the PCSP = (Z, D, C, g) where Z = set of variables, D = function which
maps every variable in Z to a finite domain, C = set of constraints and g is a function to optimize */
BEGIN
        population ← Initialisation(Z, D, g, p_size);
        REPEAT
            matepool ← Reproduction(Z, g, population, p_size, elite);
            matepool ← Crossover(Z, D, C, g, matepool, p_size, n_ospring, hc_time);
            population ← matepool;
            cycle ← cycle + 1;
        UNTIL (terminator);
END /* GAcSP */
```

```
PROCEDURE Initialisation(Z, D, g, p_size);
/* generates a p_size population of strings */
BEGIN
        FOR j = 0 to (p_size-1) DO
            population ← Generate(Z, D, g, j);
        END
        return(population);
END /* Initialisation */

PROCEDURE Generate(Z, D, g, j);
/* create string length = (fitness value+|Z| values+|Z| binary template values) */
BEGIN
        FOR i = 1 to |Z| DO
            population[j,i] ← Rand(1, | Di |);
            population[j,(i+|Z|)] ← Rand(0, 1);
            END
        population[j,0] ← Fitness(g, population, j);
        return(population);
END /* Generate */

PROCEDURE Reproduction(Z, g, population, p_size, elite);
/* biased selection of parents from the population for mating in Crossover */
BEGIN
        matepool ← Elitism(Z, g, population, p_size, elite);
        FOR j = elite to (p_size-1) DO
            matepool ← Select(Z, g, matepool, population, p_size, j);
        END
        return(matepool);
END /* Reproduction */

PROCEDURE Elitism(Z, g, population, p_size, elite);
/* selection of elite best population members for matepool */
BEGIN
        population ← Sort(population, ascending);
        FOR j = 0 to (elite-1) DO
            FOR i = 0 to (2 * |Z|) DO
                matepool[j,i] ← population[j,i];
                END
            END
return(matepool);
END /* Elitism */

PROCEDURE Select(Z, g, matepool, population, p_size, j);
/* biased selection of  population member using roulette wheel selection */
BEGIN
        FOR j = 0 to (p_size-1) DO
            p_fitness ← p_fitness + Fitness(g, population, j);
            END
        target ← Random() * p_fitness;
        member ← Rand(0, (p_size-1));
        total ← 0;
        REPEAT
            total ← total + Fitness(g, population, member);
            IF total ≤ target THEN member ← member + 1;
            IF member = p_size THEN member ← 0;
        UNTIL (total > target);
        FOR i = 0 to (2*|Z|) DO
            matepool[j,i] ← population[member,i];
```

```
            END
        return(matepool);
END /* Select */

PROCEDURE Crossover(Z, D, C, g, matepool, p_size, n_ospring, hc_time);
/* each offspring is created by exchanging parents' genetic material using their binary templates */
BEGIN
        FOR j = 1 to n_ospring DO
            matepool ← UAX(Z, g, matepool, p_size, j);
            IF C ≠ {} THENmatepool ← Repair(Z, D, C, g, matepool, (p_size- j));
            IF hc_time > 0 THEN matepool ← HC(Z, g, matepool, hc_time, (p_size-j));
            END
        return(matepool);
END /* Crossover */

PROCEDURE UAX(Z, g, matepool, p_size, j);
/* create offspring by exchanging two parents string values using their binary templates */
BEGIN
        p1 ← Rand(0, (p_size-1));
        p2 ← Rand(0, (p_size-1));
        p ← p1;
        FOR i = 1 to |Z| DO
            IF (matepool[p1,(i+|Z|)] = matepool[p2,(i+|Z|)] AND p = p1) THEN p ← p2;
            IF (matepool[p1,(i+|Z|)] = matepool[p2,(i+|Z|)] AND p = p2) THEN p ← p1;
            offspring[i] ← matepool[p,i];
            offspring[i+|Z|] ← matepool[p,(i+|Z|)];
            END
        offspring[0] ← Fitness(g, offspring, 0);
        FOR i = 0 to (2 * |Z|) DO
            matepool[(p_size-j),i] ← offspring[i];
            END
        return(matepool);
END /* UAX */

PROCEDURE Repair(Z, D, C, g, matepool, j);
/* make sure the correct number of each domain value in C are represented in offspring j */
BEGIN
        FOR i = 1 to |Z| DO
            values[matepool[j,i]] ← values[matepool[j,i]] + 1;
            END
        number_of_low_values = 0;
        FOR k = 1 to |D| DO
            IF values[k] < number of required cars of this type, as specified in C THEN
                number_of_low_values ← number_of_low_values + 1;
                under[number_of_low_values] ←k;
                END
            END
        FOR k = 1 to number_of_low_values DO
            FOR m = 1 to (number of required cars of this type -values[under[k]]) DO
                b_pos ← 0;
                FOR i = 1 to |Z| DO
                    IF values[matepool[j,i]] > number of required cars of this type THEN
                    val_over ← matepool[j,i];
                    matepool[j,i] ← under[k];
                    matepool[j,0] ← Fitness(g, matepool, j);
                    IF matepool[j,0] < b_fitness OR b_pos = 0 THEN
                        b_fitness ← matepool[j,0];
                        b_pos ← i;
```

```
                    END
                matepool[j,i] ← val_over;
                END
            END
        values[matepool[j,b_pos]] ← values[matepool[j,b_pos]] - 1;
        values[under[k]] ← values[under[k]] + 1;
        matepool[j,b_pos] ← under[k];
        matepool[j,0] ← b_fitness;
        END
    END
END /* Repair */

PROCEDURE HC(Z, g, matepool, hc_time, j);
/* for hc_time CPU seconds exchange expensive val_a's  with val_b's which reduce the offspring
fitness  */
BEGIN
    WHILE hc_time > Elapsed()
        pos_a ← Fitness(g, matepool, j);
        b_pos ← 0;
        FOR i = 1 to |Z| DO
            IF matepool[j,i] ≠ matepool[j,pos_a] THEN
                val_b ← matepool[j,i];
                matepool[j,i] ← matepool[j,pos_a];
                matepool[j,pos_a] ← val_b;
                matepool[j,0] ← Fitness(g, matepool, j);
                IF matepool[j,0] << b_fitness OR b_pos = 0 THEN
                    b_fitness ← matepool[j,0];
                    b_pos ← i;
                    END
                matepool[j,pos_a] ← matepool[j,i];
                matepool[j,i] ← val_b;
                END
            END
        val_b ← matepool[j,b_pos];
        matepool[j,b_pos] ← matepool[j,pos_a];
        matepool[j,pos_a] ← val_b;
        matepool[j,0] ← b_fitness;
        END
END /* HC */

PROCEDURE Elapsed();
/* return the number of CPU seconds elapsed */

PROCEDURE Fitness(g, population, j);
/* return the g-value (g is the objective function) of the j-th member of the population  */

PROCEDURE Fitness(g, population, j);
/* return a high cost j string value  */

PROCEDURE Rand(lower, upper);
 /* return a random number between lower and upper */

PROCEDURE Random();
 /* return a random number between 0.00 and 1.00 */

PROCEDURE Sort(population, ascending);
/* sort population in ascending fitness order */
```

Pseudo Codes for HR and Tabu:
(Z, D, C) is a CSP; in our tests, IterationLimit was set to 100,000 and TabuLengthLimit was set to 1.

```
PROCEDURE TABU_CSP_1(Z, D, C, IterationLimit, TabuLengthLimit)
BEGIN
    FOR each variable xi in Z (random order) DO
        x ← any value from Dx;
        Tabu[i] ← empty list;
        END;
    i ← 1;
    REPEAT
        S ← set of variables which label violates some constraints;
        pick a random variable y from S;
        v ← value currently assigned to y;
        V ← set of values in Dy which are not in Tabu[y];
        y ← the value which in V which involves in the least number of conflicts, break ties
              randomly;
        Make v the last element of Tabu[y];
        IF (the number of elements in Tabu[y] ≥ TabuLengthLimit)
        THEN Remove the first element from Tabu[y];
    UNTIL i ≥ IterationLimit;
END    /* of TABU_CSP_1 */
```

When TabuLengthLimit = 0, TABU_CSP_1 becomes a HR algorithm;
when TabuLengthLimit = 1, TABU_CSP_1 is a one-state Tabu algorithm.

References

Bagchi, S., Uckun, S., Miyabe, Y. & Kawamura, K., *Exploring Problem-Specific Recombination Operators for Job Shop Scheduling*, Proc., Fourth International Conference on Genetic Algorithms, 1991, 10-17

Chew, T.L., David, J.M., Nguyen, A., & Tourbier, Y., *Solving constraint satisfaction problems with simulated annealing: the car sequencing problem revisited*, Proc., International Workshop on Expert Systems & Their Applications, Avignon, France, 1992, 405-416

Cleveland, G.A. & Smith, S.F., *Using Genetic Algorithms to Schedule Flow Shop Releases*, Proc., Third International Conference on Genetic Algorithms, 1989, 160-169

Cras, J-Y., *A review of industrial constraint solving tools*, AI Perspective Series, AI Intelligence, Oxford, UK, 1993

Davenport, A., Tsang, E.P.K., Wang, C.J. & Zhu, K., *GENET: a connectionist architecture for solving constraintsatisfaction problems by iterative improvement*, Proc., 12th National Conference for Artificial Intelligence (AAAI), 1994, 325-330

Davenport, A. & Tsang, E.P.K., *Solving constraint satisfaction sequencing problems by iterative repair*, Proceeding, 14th UK Planning and Scheduling Special Interest Group Workshop, November, 1995 (to appear)

De Jong, K.A., *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Doctoral dissertation, University of Michigan, Department of Computer and Communication Sciences. Dissertation Abstracts International 36 (10), 5140B, 1975 (University Microfilms No. 76-9381)

De Jong, K. & Spears, W., *On the state of evolutionary computation*, Proc., Fifth International Conference on Genetic Algorithms, 1993, 618-623

Dincbas, M., Simonis, H., & Van Hentenryck, P., *Solving the Car Sequencing Problem in Constraint Logic Programming*, Proc., European Conference on Artificial Intelligence, 1988, 290-295

Eiben, A. E., Raue, P.E., & Ruttkay, Zs., *GA-easy and GA-hard Constraint Satisfaction Problems*, Workshop on Constraint Processing, 11th European Conference on Artificial Intelligence, 1994, 87-96

Eiben, A. E., Raue, P.E., & Ruttkay, Zs., *Solving constraint satisfaction problems using genetic algorithms*, Proc., IEEE World Conference on Computational Intelligence, 1st IEEE Conference on Evolutionary Computation, 1994, 543-547

Filipic, B., *Enhancing genetic search to scheduling a production unit*, Proc., 10th European Conference on Artificial Intelligence, 1995, 603-607

Fang H-L., Ross P. & Corne D., *A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Re-scheduling and Open-Shop Scheduling Problems*, Proc., Fifth International Conference on Genetic Algorithms, 1993, 375-382

Fleurent, C., & Ferland, J.A., *Object-oriented implementation of heuristic search methods for graph coloring, maximum clique, and satisfiability*, Trick, M.A. & Johnson, D.S. (eds.), DIMACS Series in Discrete Mathematics, 1995 (to appear)

Fox, M., & Sadeh, N., *Why is scheduling difficult, a CSP perspective*, Invited talk, European Conference on AI, 1990, 754-767

Freuder, E.C., & Mackworth, A. (eds.), *Constraint-based reasoning*, MIT Press, 1994

Freuder, E.C., & Wallace, R.J., *Partial constraint satisfaction.* Artificial Intelligence, vol 58, Nos 1-3 (Special Volume on Constraint Based Reasoning), 1992, 21-70

Freuder, E.C., Dechter, R., Ginsberg, M., Selman, B. & Tsang, E., *Systematic versus stochastic constraint satisfaction*, Proc., 14th International Joint Conference on AI, August, 1995, 2027-2032

Gent, I.P., & Walsh, T., *An empirical analysis of search in GSAT*, Journal of Artificial Intelligence Research, 1993, 47-59

Glover, F., *Tabu search Part I.* Operations Research Society of America (ORSA), Journal on Computing vol 1, 1989, 109-206

Glover, F., *Tabu search Part II.* Operations Research Society of America (ORSA), Journal on Computing vol 2, 1990, 4-32

Goldberg, D.E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Reading, MA, Addison-Wesley publishing Company Inc., 1989

Goldberg, D.E., *Real-coded Genetic algorithms, Virtual Alphabets, and Blocking*, IlliGAL Report No. 90001, Department of General Engineering, University of Illinois, September, 1990

Holland, J.H., *Adaptation in Natural and Artificial Systems*, Ann Arbor: The University of Michigan Press, 1975

Kadaba N., Nygard K.E. & Juell P.L., *Integration of Adaptive Machine Learning and Knowledge-Based Systems for Routing And Scheduling Applications*, Proc., International Expert Systems with Applications Journal vol 2, 1991, 15-27

Michalewicz Z. & Janikow C.Z., *Handling Constraints in Genetic algorithms*, Proc., Fourth International Conference on Genetic Algorithms, 1991, 151-157

Michalewicz Z., Vignaux G.A. & Groves L.J., *Genetic Algorithms for Optimization Problems*, Proc., Eleventh New Zealand Computer Conference, Wellington, New Zealand, August 16-18, 1989, 211-223

Minton, S., Johnston M.D., Philips, A.B., & Laird, P. (1990). *Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method*, Proc., Eighth National Conference on Artificial Intelligence, Menlo Park, California, USA: American Association for Artificial Intelligence. 17-24

Naden, P., *Constraint satisfaction using tabu search*. MSc Dissertation, Department of Computer Science, University of Essex, 1994

Parrello, B.D., *CAR WARS*: AI Expert , 1988, 60-64

Parrello, B.D., Kabat, W.C., & Wos, L., *Job-shop scheduling using automated reasoning: a case study of the car sequencing problem*, Journal of Automatic Reasoning, vol 2, no 1, 1986, 1-42

Richardson J.T., Palmer M.R., Liepins G.E. & Hilliard M.R., *Some Guidelines for Genetic Algorithms with Penalty Functions*, Proc., Third International Conference on Genetic Algorithms, 1989, 191-197

Schaffer, D.J., & Morishima, A., *An Adaptive Crossover Distribution Mechanism for Genetic algorithms*, Proc., Second International Conference on Genetic Algorithms, 1987, 36-40

Selman, B., Levesque, H. & Mitchell, D., *A new method for solving hard satisfiability problems*, Proc., National Conference on Artificial Intelligence (AAAI), 1992, 440-446

Selman, B. & Kautz, H., *Domain-independent extensions to GSAT: solving large structured satisfiability problems*, Proc., 13th International Joint Conference on AI, 1993, 290-295

Selman, B., Kautz, H.A. & Cohen, B., Noise strategies for improving local search, Proc., 12th National Conference for Artificial Intelligence (AAAI), 1994, 337-343

Siedlecki W. & Sklansky J., Constrained *Genetic Optimization via Dynamic Reward-Penalty Balancing and Its use in Pattern Recognition*, in Schaffer, J.D. (ed.), Proc., Third International Conference on Genetic Algorithms, 1989, 141-150

Syswerda, G., *Uniform Crossover in Genetic algorithms*, Proc., Third International Conference on Genetic Algorithms, 1989, 2-9

Tsang, E.P.K., & Warwick, T., *Applying Genetic algorithms to Constraint Satisfaction Optimisation Problems*, Proc., European Conference on Artificial Intelligence, Stockholm, Sweden, 1990, 649-654

Tsang, E.P.K., *Foundations of Constraint Satisfaction*, Academic Press, London and San Diego, 1993

Wallace, R.J., & Freuder, E.C., *Conjunctive width heuristics for maximal constraint satisfaction*. Proc., National Conference on Artificial Intelligence (AAAI), 1993, 762-768

Warwick, T., & Tsang, E.P.K, *Using a Genetic Algorithm to Tackle the Processors Configuration Problem*, Proc., ACM Symposium on Applied Computing, 1993, 217-221

Warwick, T., *A GA Approach to constraint satisfaction problems*, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK, 1995

Whitley, D., Starkweather, T. and Shaner, D., *The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination*, in Handbook of Genetic Algorithms (ed. L. Davis), New York, USA: Van Nostrand Reinhold, 1991, 350-372

Zhu, K., 1993, Unpublished results in the GENET project, EPSERC funded (UK), reference GR/H75275, (Dr. Zhu was a visiting scholar at the University of Essex in April - November 1993)