

A GA Approach To Constraint Satisfaction Problems

T. J. Warwick

A thesis submitted for the degree of Ph.D.

**Department of Computer Science
University of Essex
Wivenhoe Park
Colchester
Essex
CO4 3SQ**

1995

Abstract

Genetic algorithms (GAs) are generally considered to be unsuitable for constraint based problems, particularly those with tight constraints. Some researchers have developed specialised techniques for solving specific groups of problems. In this research we contribute to this work by developing a flexible generic GA which can exploit problem constraints. The GA has been designed to tackle and exploit an important class of optimisable constraint based problems, namely partial constraint satisfaction problems (PCSPs). This new GA is a strategy which incorporates an adaptive template type crossover and hill-climbing component (HC). This GA strategy which we call GAcSP, combines the robust global power of the GA with the specialist power of the HC to form a powerful combination - the GA finds the hills and the HC climbs them. A crossover operator has been developed which has a learning capacity which can exploit problem constraints. The ability of GAcSP is demonstrated by tackling two distinct problems, namely the processors configuration problem and the car sequencing problem. Both problems are *NP*-hard, and represent a serious challenge to GAcSP. Results from these experiments show that GAcSP can out-perform specialised approaches, is not deterred by problem size, and not limited to tackling solvable problems only. The GAcSP strategy provides an effective tool for tackling a class of PCSPs. The power of GAcSP is due to the optimal balance of work between GA and HC by exploiting the abilities of both components. This synergistic combination between GA and HC gives the GAcSP flexible powers in tackling larger problems.

CONTENTS

Acknowledgements	iv
Chapter 1 Introduction	1
1.1 Project Motivation And Objective	1
1.2 Introduction	2
1.3 What Is PCSP ?	3
1.3.1 Partial Constraint Satisfaction Problems (PCSPs)	3
1.3.1.1 PCSP Introduction	3
1.3.1.2 PCSP Features	4
1.4 What Is A GA	5
1.4.1 GA Background	5
1.4.1.1 Introduction	5
1.4.1.2 Origin Of The GA	5
1.4.1.3 Exploitation Versus Exploration	6
1.4.1.4 Canonical GA	6
1.4.2 The Schemata Theorem Of GAs	7
1.4.3 Representations	13
1.4.3.1 Why Binary Representations ?	13
1.4.3.2 Why Large Alphabets ?	14
1.4.3.3 Real-Coded Theory	16
1.4.3.4 Alternative Schemata Theorem	17
1.4.4 GA And Constraints	18
1.4.4.1 Introduction	18
1.4.4.2 Penalty Function	20
1.4.4.3 Specialist Crossover Operators	22
1.4.4.4 Specialist Mutation Operators	26
1.4.4.5 Inversion Operators	26
1.5 Summary	27
Chapter 2 Design Of A New GA For Solving PCSPs	32
2.1 Analysis Of Applying GA To PCSP	32
2.1.1 Control Strategy	32
2.1.2 Representation	33
2.1.2.1 Binary Representation	34
2.1.2.2 Real-Coded Representation	36
2.1.3 Exploiting Constraints	37
2.1.3.1 Hard Constraints	38
2.1.3.2 Soft Constraints	39
2.1.4 Crossover Design Analysis	39
2.1.4.1 Crossover Operator	39
2.1.5 Mutation Design Analysis	42
2.1.5.1 Repairer	42
2.1.5.2 Hill-Climber	42
2.1.6 Population Dynamics Analysis	43

2.1.7	A GA System	45
2.2	What Is GAcSP ?	45
2.2.1	Outline Of GAcSP	45
2.2.2	PCSP Representation Framework	46
2.2.3	Objective Function	47
2.2.4	GAcSP Initialisation Operator	48
2.2.4.1	Initialisation Function	48
2.2.5	GAcSP Reproduction Operator	48
2.2.5.1	Elitism Function	49
2.2.5.2	Reproduction Function	49
2.2.6	GAcSP Crossover Operator	49
2.2.6.1	UAX Function	50
2.2.6.2	Greedy Repair Function	52
2.2.6.3	Optional Hill-Climber Function	53
2.2.7	GAcSP Parameters	55
2.2.8	Population Dynamics	56
2.2.8.1	External Dynamics	56
2.3	Summary	57
Chapter 3	Tested Domains	60
3.1	The Processors Configuration Problem (PCP)	60
3.1.1	Definition	60
3.1.2	Graph Theory	61
3.1.3	Representation	65
3.1.4	Objective Function	66
3.2	The Car Sequencing Problem (CarSP)	67
3.2.1	Definition	67
3.2.2	Example Of A Solvable CarSP	70
3.2.3	The Penalty Function	71
3.2.3.1	Basic Method	71
3.2.4	Theoretical Lower Bound For Unsolvable CarSPs	75
3.2.5	Representation	76
3.2.6	Objective Function	76
3.3	Summary	77
Chapter 4	Implementation And Testing Methodology	80
4.1	Implementation Details	80
4.1.1	Design Decisions	80
4.1.2	Compiling And Running Programs	80
4.2	Program GA1	81
4.2.1	Description	81
4.2.2	Domain Specific Functions	81
4.2.2.1	PCP Evaluation Functions	83
4.2.2.2	CarSP Evaluation Functions	83
4.2.3	Program Parameters	83
4.2.4	Population Dynamic	83
4.3	Experimental Methodology	84

4.3.1	Recording Results	84
4.3.2	Terminating Conditions	84
4.3.2.1	GA Conditions	84
4.3.2.2	HC Conditions	85
4.3.3	Methods Of Analysis	85
4.4	Summary	85
Chapter 5	PCP Results	87
5.1	Plan	87
5.2	GAcSP Tackling Different Sized PCPs Without HC	87
5.2.1	Results Of Experiment 5.2	88
5.2.2	Experiment 5.2 Discussion	89
5.3	GAcSP Tackling Different Sized PCPs With HC	94
5.3.1	Results Of Experiment 5.3	94
5.3.2	Experiment 5.3 Discussion	95
5.4	Summary	97
Chapter 6	CarSP Results	99
6.1	Plan	99
6.2	GAcSP Tackling CarSPs Of Different Tightness	100
6.2.1	Results Of Experiment 6.2	101
6.2.2	Experiment 6.2 Discussion	102
6.3	GAcSP Tackling CarSPs Of Different Size	110
6.3.1	Results Of Experiment 6.3	110
6.3.2	Experiment 6.3 Discussion	111
6.4	GAcSP Tackling Unsolvable CarSPs	114
6.4.1	Results Of Experiment 6.4	115
6.4.2	Experiment 6.4 Discussion	117
6.5	Summary	125
Chapter 7	Conclusion And Future Development	129
7.1	Contribution Of This Research	129
7.2	Further Developments	130
7.2.1	Using Diversity Information	130
7.2.2	Tackling Binary CSPs	134
7.3	Summary and Conclusion	134
	Summary Of Important Symbols	137
	Appendix A HR and TABU Pseudo Code	138
	Bibliography	139
	Definition Index	154

Acknowledgements

To thank everyone who motivated, helped, or inspired me would require this thesis to be nothing other than a list of names. So I thank certain people for their special help. To the rest who cannot be mentioned, they will not be forgotten.

I could not have found a better ally and guide in the Ph.D. journey than my supervisor Dr. Edward Tsang. His patience, humour, and intellectual integrity was a constant source of inspiration and motivation.

To all my family and friends, especially my parents who believed in me more than I did myself, I shall always love them.

The financial support of the Science and Engineering Research Council should be mentioned, without which this thesis could not have started, survived or finished.

Finally, I dedicate this work to someone special, I hope one day to meet.

Chapter 1 Introduction

1.1 Project Motivation And Objective

Constraint satisfaction problems (CSPs) can be found in many application areas of AI - instances include boolean satisfiability, scene labeling, graph isomorphism, scheduling to name a few. Research into CSPs is motivated by the need to provide efficient methods to tackle them. In CSPs the requirement is to find a single solution, a number of solutions or all solutions which satisfy the constraints. In many applications the requirement is not just to find solutions but to find an optimal solution, where optimality is measured by some application specific cost function. Resource allocation in scheduling is an optimisable constraint satisfaction problem (CSOP) which requires the allocation of resources to jobs or machines to jobs in the most cost effective way. However, in general, CSOPs are *NP*-hard and techniques used to tackle them suffer from combinatorial explosion (i.e. search space grows exponentially). The size of the search space in combinatorial problems can prevent conventional general search methods (which lack focus) from being useful, due to the time taken to find solutions. CSOPs can be considered to belong to a category of CSPs where all solutions are required, because in principle cost function values of all solution tuples must be compared. With tightly constrained CSOPs *problem reduction* can be used to prune off parts of the search space. Loosely constrained CSOPs are likely to be more difficult because less of the search space can be pruned, therefore the number of possible solutions could be larger.

A *Branch and Bound* (B&B) algorithm with a good heuristic which can give an accurate estimation of the cost function could be used, but would be unlikely to be able to solve very large problems [Tsang, 1993]. Also heuristics are domain specific, and good heuristics are sometimes difficult or expensive to find. We believe stochastic search can provide a useful alternative to methods such as B&B in tackling combinatorial CSOPs, where optimality can be sacrificed for speed. Stochastic search is a class of search methods which have an element of randomness and use heuristics to guide the search from one point in the search space to another. The Genetic Algorithm (GA) is a robust stochastic algorithm, which has been found to be successful in combinatorial search [Mühlenbein, 1989; Cleveland and Smith, 1989]. Earlier

work has also found that GA is a promising approach to tackling loosely constrained CSOPs [Tsang and Warwick, 1989].

Furthermore, in "real-world" problems, complex constraints cannot always be completely satisfied. These unsolvable optimisation problems form a class of problems which we refer to as partial constraint satisfaction problems (PCSPs), where the objective is to violate as few constraints as possible. We are primarily motivated by the need for an efficient flexible search strategy for tackling scheduling problems. Our objective in this research is to design and empirically test a generic search strategy based on stochastic search which can tackle a class of PCSPs, which includes solvable and unsolvable CSOPs. We believe a successful approach can be achieved by the combination of a robust GA with local improvement which will provide a strategy with the ability to efficiently exploit PCSPs.

1.2 Introduction

Genetic algorithms are generally considered to be unsuitable for constraint based problems, particularly those with tight constraints [Goldberg, 1989]. Some researchers have developed techniques for solving specific groups of problems with numerical constraints [Michalewicz et al., 1989; Richardson et al., 1989; Siedlecki and Sklansky, 1989]. Our goal in this research is to design a flexible GA, to tackle an important class of constraint based problems, namely PCSPs. Research has already indicated that GAs could provide a useful approach for tackling CSOPs [Tsang and Warwick, 1989]. In order to exploit the features of PCSPs, a new GA strategy has been designed which incorporates a template type crossover and hill-climbing component (HC). This GA strategy which we call GAcSP, combines the robust global power of the GA with the specialist power of the HC to form a powerful combination - the GA provides HC with promising search space points. The HC component can exploit domain specific knowledge without compromising its generic ability. A crossover operator has been developed which has a learning capacity which can exploit problem constraints. The ability of GAcSP is demonstrated by tackling two distinct problems, namely the processors configuration problem and the car sequencing problem. Both problems are *NP*-hard, and represent a serious challenge to GAcSP. Results from experiments show that GAcSP can out-perform specialised approaches, is not deterred by

problem size, and not limited to tackling solvable problems only. The GAcSP strategy provides an effective tool for tackling a class of PCSPs. The power of GAcSP is due to the optimal balance of work between GA and HC which exploits the abilities of each component. This synergistic combination between GA and HC gives the GAcSP flexible powers in tackling larger problems.

1.3 What Is PCSP ?

1.3.1 Partial Constraint Satisfaction Problems (PCSPs)

1.3.1.1 PCSP Introduction

Many artificial intelligence and computer science problems are instances of CSPs [Tsang, 1993]. These include scene labeling, graph isomorphism, boolean satisfiability, graph colouring and scheduling (especially resource allocation). The CSP is known to be *NP*-complete and requires heuristic techniques to solve it. The efficiency of CSP solving techniques can be improved by using heuristics to guide the process [Meseguer, 1989]. In this research we use a heuristic approach assisted by local improvement techniques.

The CSP has a finite set of variables, each variable has a finite domain of values and there is a finite set of constraints. A solution tuple is an assignment of a value to each variable (from their respective domains) satisfying the constraints. The PCSP is an optimisation problem, where a solution tuple is an assignment of any value from each variable domain and an objective function g can be defined which maps every solution tuple to a numeric value. The PCSP with a solution tuple satisfying the constraints is equivalent to CSP. Otherwise, the requirement for PCSPs where the constraints cannot be satisfied is to find *the best* solution tuple which minimises or maximises the objective function. If there is a solution tuple for a PCSP which satisfies the constraints then it is equivalent to a CSP.

° **Definition 1.1 (CSP)** A constraint satisfaction problem (CSP) is a triple:

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

where Z = set of variables $\{x_1, x_2, \dots, x_N\}$

D = set of discrete domains for each variable in Z $\{v_1, v_2, \dots, v_N\}$

and C = set of constraints on arbitrary subsets of variables in Z , restricting the values that they can take together.

Each discrete variable x_i has a domain $v_i = \{v_{i1}, v_{i2}, \dots, v_{ik}\}$ where the cardinality $k = |v_i|$.

If in addition to the CSP, we include the requirement that g is a function which maps every solution tuple to a numeric value we have a PCSP, formally defined as:

° **Definition 1.2 (PCSP)** A partial constraint satisfaction problem (PCSP) is a quadruple:

$$(Z, D, C, g).$$

1.3.1.2 PCSP Features

We can identify important features for problems formalised as PCSPs. These features can be exploited by specialised algorithms which include the GAcSP. These main features are:

° **Definition 1.3 (Search Space)** The GA *search space* is a network, where nodes are solution states and arcs the means to move between solutions.

(1) The PCSP search space is finite.

(2) If the variables are ordered then the search space is a tree with a depth equal to the number of variables and the width at a level, is equal to the domain size of the corresponding variable.

(3) Sub-trees may have similar depth and width.

(4) Because constraints represent relationships between variables, the effect of assigning a value to one variable can be propagated to others.

(5) Constraint relationships can lead to feasible and infeasible regions in a search space.

These features are used to motivate the design of the GAcSP strategy as a specialised algorithm

for PCSPs. We shall be exploring the GAcSP design issue in Section 2.1.

1.4 What Is A GA

1.4.1 GA Background

1.4.1.1 Introduction

GAs may out-perform both specialised and random search methods on complex search spaces [De Jong, 1975]. They efficiently exploit representation information to speculate on new search points with expected improved performance, and differ from other optimisation methods by working on parameter coding not parameters themselves, searching from a population of search points, using objective function information not domain specific knowledge, and using probabilistic rules for moving from one state to another state rather than deterministic.

1.4.1.2 Origin Of The GA

John Holland established the field of GAs with the publication in 1975 of *Adaptation in Natural and Artificial Systems*. The fundamental contribution was the use of binary strings to represent complex structures and the application of transformations to improve them. The GA could evolve a population of binary strings by simple, yet powerful, syntactic actions upon them. With certain search space conditions the GA would tend to converge on solutions which were at or close to the global optimum. Holland recognised however that non-linearity or false peak (i.e. *epistasis*) represented an obstacle to adaptation.

Holland's goals were to abstract and explain the adaptive processes of natural systems and design artificial systems software that used these mechanisms. One important theme of subsequent research has been robustness, where GAs was to provide effective, efficient search across a broad range of problems, and not be limited by restrictive assumptions about the search space (assumptions concerning continuity, derivatives, unimodality etc.). In order to provide effective and efficient search, the balance between discovering new knowledge and exploiting what is known must be controlled.

1.4.1.3 Exploitation Versus Exploration

There is a conflict between exploiting what is known and obtaining new information, where both activities cannot be undertaken simultaneously (e.g. q -armed bandit problem). That is, trials which only exploit the observed best, can result in perpetuating an error. Or trials which are only allocated to reducing error, can result in a loss of performance. GAs must balance the need to exploit the observed best and the need to explore which maintains an optimal performance. Holland has shown [Holland, 1973; 1975] that a GA can maintain a near optimal balance between robustly searching the space to discover better solutions (exploration) (e.g. a pure robust method is random search), and using current but localised knowledge as in hill climbing (exploitation). A GA is based on the simple heuristic that the best solutions are found in regions of the search space containing good solutions and that these regions can be identified and sampled. We consider the basis for this important assumption in analysing how a canonical or simple GA works.

1.4.1.4 Canonical GA

The canonical [Schaffer, 1987] or simple GA [Goldberg, 1989] (SGA) developed from Holland's work, has three main operators reproduction, crossover and mutation. This simple model of GA has the following assumptions:

- **Definition 1.4 (Binary string S_{bin})** Binary strings may be represented symbolically as a string S_{bin} , composed of l binary integers, $S_{\text{bin}} = b_1, b_2, \dots, b_l$, where $b_i \in \{0, 1\}$ for $i = 1, 2, \dots, l$.
- (a) Binary strings are fixed length.
- **Definition 1.5 (String population $P(t)$)** At time (or generation) t there is a population $P(t)$ of n individual strings S_{bin}^p , where $p = 1, 2, \dots, n$.
- (b) A finite dynamic population of strings as a database of points representing what is known about the search space.

- **Definition 1.6 (String fitness)** The *fitness* of a string is the optimisation function value referred to in GAs as the evaluation function or fitness function.
- (c) Each string has a relative ability (*fitness*) to survive and produce offspring.

The fact that GAs make few assumptions about their problem domain (weak) make them useful to a broad range of problems. GAs can be a powerful, broad based method using local information. However, they may not provide a superior performance to specialised techniques which exploit available domain knowledge. Opportunities do exist to use problem specific knowledge in the GA operators but at the cost of becoming domain specific.

- **Definition 1.7 (Non-overlapping)** A non-overlapping population is where the number of offspring generated each GA cycle is equal to the population size.

Figure 1.1 outlines the control flow of the SGA and is described in the following details. During initialisation, a random population of binary strings is generated. The three operators reproduction, crossover and mutation act upon this population of strings for each cycle (or generation) of the GA. Reproduction generates a mating pool *matepool* of potential parents by selecting strings from the population with a bias towards fitter strings (low fitness for minimisation problems). The first stage of the one-point crossover operator randomly selects two parent strings from the *matepool*. For the second stage the parent strings are cut at a randomly selected point and corresponding sections from each parent are exchanged, generating two offspring. A mutation operator is used with a low probability to change an offspring string element. The process of selecting parents and creating offspring continues until a new population is generated and is described as *non-overlapping* or *generational*. The *Schemata Theorem* of GAs allows us to quantify and predict the behaviour of the SGA.

1.4.2 The Schemata Theorem Of GAs

Holland [1975] recognised an important feature from biology - that, an individual can influence the future development of a population by surviving to reproduce. In our artificial setting we can quantify this notion of survival using the Schemata Theorem (or *Fundamental Theorem*) presented formally below:

- **Definition 1.8 (Schema)** A schema H describes a subset of strings with similarities at certain string positions.

The power of the GA comes from the processing of string similarities (templates), called schemata (singular schema). We can define a schema H over the alphabet $\{0, 1, \#\}$, where the metasympol "#" is a pattern matching device which can match 0 or 1. For example, the length $l = 4$ schema $H = \#1\#0$, describes a subset of strings $\{0100, 0110, 1100, 1110\}$. Where the subset of strings have similarities at positions 2 ($b_2 = 1$) and 4 ($b_4 = 0$). Furthermore, the fitness $f(H)$ for schema H is the average fitness of the subset of strings it describes.

For binary strings of length l there are $(k+1)^l$ schemata, where alphabet valency $k = |\{0, 1\}|$. The number of *unique* schemata in a binary string is 2^l and depending upon diversity the number in a n string population will be between 2^l and $n \cdot 2^l$.

Two properties of schema are *schema order* and *defining length*.

- **Definition 1.9 (Schema Order)** The order $o(H)$ of a schema H , is the number of fixed positions (i.e. 0's or 1's) in H . For example, $o(\#1\#0\#) = 2$.
- **Definition 1.10 (Schema Defining Length)** The defining length $\delta(H)$ of a schema H is the distance between the first and last fixed positions. For example, $\delta(\#1\#0\#) = 2$.

At time t there are m examples of schema H contained in population $P(t)$. During reproduction a string $S^{\text{p}}_{\text{bin}}$ is copied according to its fitness f_p with probability $\rho_p = f_p / f_{\text{avg}}$, where

$$f_{\text{avg}} = \left(\sum_{p=1}^n f_p \right) / n. \quad (1.1)$$

After picking a non-overlapping population of size n with replacement from the population $P(t)$, we expect to have $m(H, t+1)$ representatives of the schema H in the population at time $t+1$ according to the reproductive schema growth equation:

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{f_{\text{avg}}}. \quad (1.2)$$

Schemata growth depends upon the ratio of schema fitness to the population average fitness. Above average schemata will generate an exponentially increasing number of copies with those below average decreasing. The survival of schemata also depend upon the disruptive effects of crossover and mutation.

Consider the second stage of the one-point crossover operator which randomly selects a point at which to cut the parent strings. There are $l - 1$ possible crossover sites for selection. The probability that a schema H is disrupted will depend upon the order $o(H)$ and the defining length $\delta(H)$. The more fixed positions and the further apart they are, the more likely a schema will be disrupted. An estimate of schema H survival is where it is destroyed with probability $\rho_d = (\delta(H) / (l - 1))$ and survives with probability $\rho_s = (1 - \rho_d)$. Generally, a lower bound for schema H surviving crossover, where ρ_c is the probability of crossover being performed, is

$$\rho_s \geq 1 - \rho_c \cdot \frac{\delta(H)}{l - 1} \quad (1.3)$$

More accurate estimates depend upon whether crossover is carried out between identical or complementary strings. Also, *string gains* [Bridges and Goldberg, 1987] can result from crossover, where new schemata can be created.

With mutation, each string value survives with probability $(1 - \rho_m)$, where ρ_m is the mutation probability. Each string value is independent for each of the $o(H)$ fixed positions the schema survives with probability $(1 - \rho_m)^{o(H)}$. For small values of $\rho_m < 1$ the schema survival probability can be approximated by $1 - o(H) \cdot \rho_m$. [Goldberg, 1989]

The crossover survival rate and the disruptive effect of mutation can be combined to give an estimate for the survival of schema H due to these operators as

$$\rho_s \geq 1 - \rho_c \cdot \frac{\delta(H)}{l - 1} - o(H)\rho_m \quad (1.4)$$

To obtain a more accurate estimate for the growth of schema H , taking into account the disruptive effect of crossover and mutation, we have the Schemata Theorem or Fundamental

Theorem of GAs:

Theorem 1.1 (Schemata Theorem)

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{f_{\text{avg}}} \cdot \left(\frac{\delta(H)}{1 - \rho_c \cdot l} - (1 - \rho(H) \cdot \rho_m) \right). \quad (1.5)$$

- **Definition 1.11 (Building Blocks)** Building blocks are high fitness schemata with short defining length. [Goldberg, 1989]

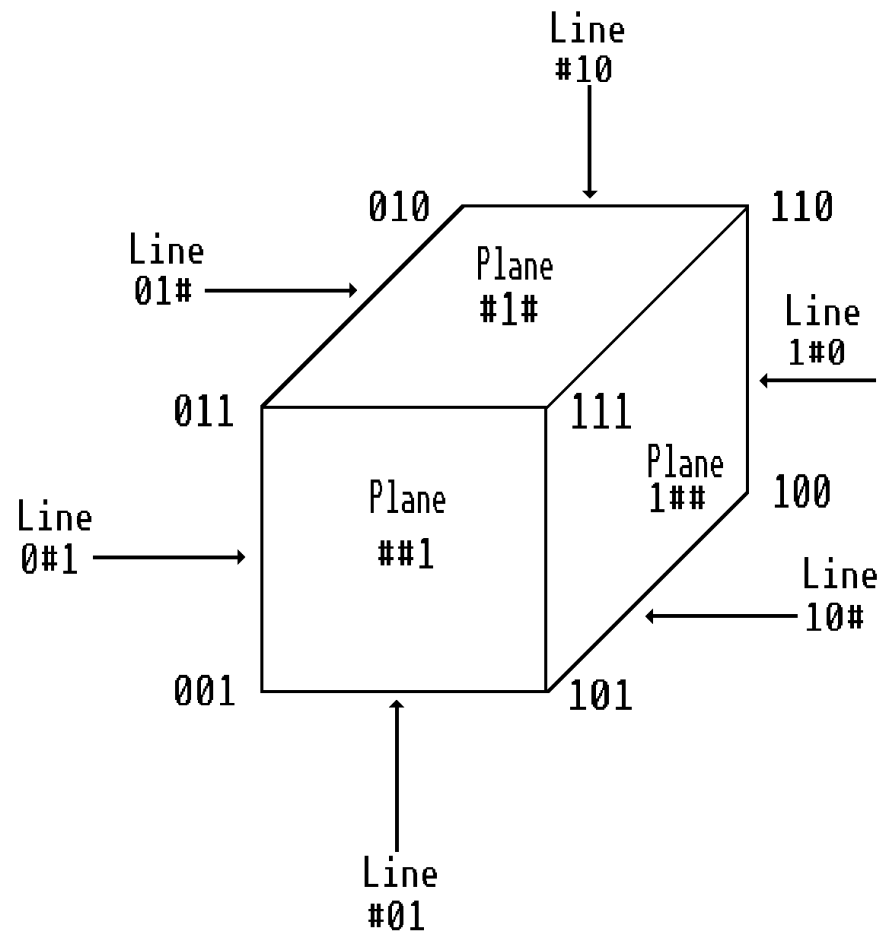
We can use the Schemata Theorem to predict that short high fitness schemata will increase exponentially, from one generation to the next. These schemata are important *building blocks* in the search for optimal solutions.

- **Definition 1.12 (Implicit Parallelism)** Implicit parallelism is the parallel processing of schemata, estimated by Holland as $O(n^3)$ for a population of n strings. [Goldberg, 1989]

This exponential increase of building blocks is termed *implicit parallelism* and the GA effectively samples from the set of hyperplanes in proportion to its observed fitness in relation to the population average. Holland's [1975] $O(n^3)$ schemata processing estimate, means that through the processing of only n string structures each generation, a GA processes something like n^3 schemata.

Pioneering work by Bagley [1967] and Rosenberg [1967] support the claim of building block hypothesis by empirical work. The development of Walsh functions by Bethke [1981] has given an efficient analytical method for determining average schemata fitness values.

The notion of the hypercube and hyperplanes is a useful aid to visualise the mathematical principles behind implicit parallelism. Consider the hypercube in Figure 1.2, which is a three-dimensional vector space for $l = 3$ length binary strings and schemata. The corners of the hypercube represent search space points or order 3 schemata. Hypercube planes represent order 1 schemata and lines order 2 schemata. According to the Schemata Theorem, given an GA processing a population of binary strings, the GA is processing hyperplane schemata information. This result can be generalised to an l -dimensioned hyperplane binary search space.

Figure 1.2: Hypercube for length $l = 3$ binary string

Although, the Schemata Theorem supports the idea of the growth of above average, short defining length schemata, it does not provide details on how this assists the GA in finding optimal solutions. The Schemata Theorem would suggest that long, highly fit schemata are on a decline in a population due to the crossover and selection mechanism. It is conjectured [Whitley and Kauth, 1989], that a threshold phenomena occurs, when the number of similar short schemata reach a critical mass in the population, at this point the probability of creating long schema of interest will become dominant. Moreover, if the short schemata accurately reflect the optimisation of the search space, optimal long schemata will emerge, otherwise peak to a sub-optimal optimum.

1.4.3 Representations

1.4.3.1 Why Binary Representations ?

° **Definition 1.13 (Binary-coded)** Binary-coded representations are where the fundamental elements which comprise the representation are selected from the alphabet $\{0, 1\}$.

As we have already seen, the use of binary representations is supported by the Schemata Theorem and the simple GA. The binary string was accepted early on as the best way to maximise the implicit parallelism inherent in the GA. Goldberg's *principle of minimal alphabets* (given below) encouraged this view. There are a number of reasons why binary representations have dominated GA research including the simple analysis of binary vectors, the elegance of GA operators, and the requirements of computational speed.

Holland contends that a representation should have the following characteristics:

- (1) Representation elements are not position dependent.
- (2) There is a small number of values at each position.
- (3) The representation can define sufficient schemata.

Characteristics (1) and (2) have been demonstrated analytically and empirically for binary strings [Holland, 1975; Bethke, 1980; Goldberg, 1989]. Characteristic (3) ensures the representation does not restrict the search, by making sure it is not prevented from accessing any part of the search space (i.e. complete search).

Goldberg's [1989] principles of *minimal alphabets* for representation designers follow on from, and support, Holland's. The first principal of *meaningful building blocks* suggests that a coding should be selected, so that short, low-order schemata are relevant to the underlying problem and relatively unrelated to schemata over other fixed positions. When using binary strings this principal can be checked using Walsh transforms, which map the binary string to a real parameter space. However it may not be practical to do so, depending on the size of the problem, where there will be 2^l coefficients to calculate. The second principal of *minimal alphabets* suggest that the smallest alphabet should be selected that permits a natural expression of the problem.

The Schemata Theorem suggests that small alphabets maximise the number of schemata available for GA processing. The maximum number of schemata Max_{Sc} per bit of information in a string coded over an alphabet of cardinality k may be calculated [Goldberg, 1989] as:

$$Max_{Sc} = \sqrt[k]{k + 1}. \quad (1.6)$$

Maximising this expression with respect to k and holding the constraint that alphabets must be cardinality 2 or greater, the optimal cardinality is $k = 2$. [Goldberg, 1989]

We can understand the difference between binary and real-coded (i.e. non-binary) alphabets in an intuitive way by recognising there are more possible hypotheses in a string using binary alphabets. For example, 101 in binary rather than a real-coded value 5, because the fitness could be due to the middle 0 or the first 1 in the binary.

1.4.3.2 Why Large Alphabets ?

- **Definition 1.14 (Real-coded)** Real-coded representations are constructed from fundamental elements other than the binary alphabet $\{0, 1\}$.

The recent impetus for the use of real-coded representations arose from Weinburg's 1970 thesis. (Earlier research carried out by Selfridge [1959], and Friedman [1959] used real-coded genes in an adaptive context.) Research that followed, by Bosworth, Foo, and Zeigler [1972], and Bethke [1981] did not attempt to incorporate Holland's Schemata Theory. Empirical evidence to support the use of real-coded representations may have attracted the interest of other researchers to use GAs. The reluctance by the AI community to embrace this may be partly due to the fact that much of GA research has considered only simple binary representations, whereas most AI research uses more complex representations. There is a trade-off between the speed of binary string GA applications and slower GAs based on more complex representations.

- **Definition 1.15 (Traveling Salesman Problem (TSP))** An *NP*-complete problem with the task of finding the shortest distance between N cities, visiting each once and ending at the starting point.

There are many problems which cannot be expressed in terms of a simple binary alphabet, and still be amenable to the standard genetic operators. There are two main reasons. Since

the number of possible actual interpretations is unlikely to be a power of two, some interpretations will be redundant, weighting the possible choices unevenly. Also, some mutations are extremely unlikely (e.g. *hamming cliffs*, see (b) below). Many problems therefore require more sophisticated data structures and GA operator modifications. The data structure is intimately involved in controlling the behaviour of the task. The difficulty with simple linearising data structures, mapping them (encode) into the GA string representation, then reverse the process (decode) to produce new structures can be seen with representation issues in the Travelling Salesman Problem (TSP) [Goldberg and Lingle, 1985; Grefenstette et al., 1985; Oliver et al., 1987; Whitley et al., 1989].

- ° **Definition 1.16 (Hamming Cliffs)** Hamming cliffs are where the binary representations for adjacent integers differ in every fixed position. For example, integers 31 and 32, and their corresponding binary representations 01111 and 10000.

Possible reasons for using real-coding are: (1) "Comfort" with one gene to one variable correspondence - This is more psychological than technical [Goldberg, 1990]. (2) Avoidance of *hamming cliffs* and other artifacts of mutation - Binary coded GAs can be stopped from reaching a points in the search space by hamming cliffs. Consider the problem where the optima is 100, and schema fitness $f(0\#\#) > f(1\#\#)$, then the part of the search space for binary strings with 0 in the first position will be above average causing the GA to converge to say 011. Although 011 is close to the optima 100 it needs three bit mutations to reach it, and this is unlikely (i.e. $O(p_m^3)$). Real-coded GAs using special operators such as the *adjacency mutation operator* of Davies and Coombs [1987], can *creep* around the search space by successive mutations. For example using a real-coded schema, after convergence to a value, successive mutation could correct the solutions until the optimum is reached.

- ° **Definition 1.17 (Gray Codes)** Gray codes are binary strings where the representations of adjacent integers differ by only a single bit.

The use of *gray codes* has been suggested to overcome the problem of hamming cliffs [Bethke, 1981] but introduces high order non-linearities with respect to recombination. Goldberg [1990] suggests a solution to overcome the high order non-linearities, using both a simple bit change

and decision-dependent -change mutation operators. (3) Real-coded alphabets converge more quickly than small coded ones, but the quality of the solution can degrade with increasing alphabet cardinality k [Goldberg, 1990]. (4) Real-coded alphabets reduce the combinatorial dimension of the problem, which in turn reduces the opportunity for above average fitness schemata to exist which leads the GA away from optimal solutions (*deception*) [Goldberg, 1990]. Although there is empirical work to support the use of real-coded GAs for tackling problems, theoretical work is important in analysing this success.

1.4.3.3 Real-Coded Theory

There are problems in extending the Schemata Theorem to real-coded and position dependent representations. (We consider some of these problems in the next section.) The most important aspect of the search space to the Schemata Theorem is the "type" of building block available. The analysis of representation space can be developed by investigating building blocks that compose a schema and the presence of hyperplanes. It is the relationships between the problem parameters defines the hyperspace. The GA success in finding optimal solutions will depend upon these schema correlating with performance. Analysis of schemata is important because the GA sees the problem through the representation. The Walsh function analysis devised by Bethke [1981]; ([Holland, 1987], extended by Holland to include non-uniform populations [Goldberg, 1989]) can be used to measure this correlation for binary representations. Mason [1991] has extended the Walsh function analysis for real-coded alphabets, including the analysis and construction of deceptive problems where the schemata lead the GA away from the optimum. Goldberg [1990] has presented the notion of *virtual alphabets* to support a theory of operators for real-coded GAs, where virtual alphabets are constructed from virtual characters which are composed of problem variables. So, if the GA is tackling a problem using a high cardinality alphabet problem variables will compete in the form of schemata.

Also, in support of real-coded representations, Antonisse [1989] has questioned the analytic framework used to justify minimal alphabets (see Section 1.4.2). The two critical issues are concerned with how the symbolic encoding of the search space effects the representational power of the search, and what legal structures can be encoded. Earlier, in Section 1.4.2 we used the "#" symbol in order to demonstrate the power of GA processing in terms of pattern matching.

Antonisse reinterprets the use of the "#" symbol in analysing schemata where the class of strings is a single subset extending it into the case where every possible subset of individuals is counted as a schema. Using the example string of length 4 with alphabet {0, 1, 2} Holland's interpretation of the schema "000#" refers to the set of schemata {0000, 0001, 0002}, whilst Antonisse's leads to {0000, 0001}, {0000, 0002}, {0001, 0002}, and {0000, 0001, 0002}. In Section 1.4.2 the analysis suggested that the individual binary string schemata processing is maximised when $k = 2$ (Equation 1.6). However if we consider Antonisse's interpretation, the counting argument to express all schemata is not $k + 1$ but $2^{(k-1)}$ [Antonisse, 1989]. Therefore, the more expressive real-coded alphabet is seen to carry much more power, providing *finer-grained* tools for the construction of adaptive plans.

The importance such work presents to the GA community is not so much in the 'overthrow' of established GA theory but to deepen and widen the issue of GA analysis. Just how valid schema counting comparisons are, for different coding schemes, has been questioned. Antonisse [1989] admits that some experiments in practice do not support this idea. This occurs when all the binary encodings are valid but not all higher cardinality alphabets can be decoded by the objective function. This suggests that the encoding should contain no redundant or un-decodable information. Grefenstette [GA-Digest vol 5, issue 19, 1/8/1991] has argued that a mis-interpretation of the Schemata Theorem leads to a strong building block hypothesis which highlights a limitation when applying the q -armed bandits analogy to schema processing. Aware of this difficulty, Vose and Liepins [1991] developed an alternative Schemata Theorem framework for measuring the effectiveness of crossover operators and disruption rates of *generalised* schemata, which they call *predicates*.

1.4.3.4 Alternative Schemata Theorem

The Schemata Theorem provides an explanation for the success of an SGA using a binary representation but does not explain the success for GAs using different operators and non-binary representations. Conversely, there are a range of problems which are not well optimised by GAs. Such problems are considered *GA difficult* [Vose and Liepins, 1991] for a number of reasons, among which are deceptiveness, sampling error, and schema disruption. Deceptiveness occurs when above average schemata do not lead the GA towards the optimum. Sampling error

occurs when schema with above average utility but below average fitness, with respect to the current population are forced to die out due to reproductive pressure. Schema disruption occurs when the crossover operator prevents the progression from low order to high order schemata. To explain the success and failure of GAs requires an alternative schema analysis.

An alternative interpretation of the Schemata Theorem has been outlined by Vose [1991]. This re-interpretation emphasises the importance of schemata disruption on what building blocks can be formed from the crossover schemata interaction. Vose [1991] regards schemata as predicates which map binary strings into the set {true, false}. Using predicates instead of schemata allows a more generalised view of schemata interaction. New concepts of *locality*, *globality*, *monotonicity*, and *stability* are defined. Local predicate building blocks important to the formation of the optimum can have a fitness less than a population average. Global predicate building blocks always have, a fitness greater or equal to *any* given population average. A predicate is only global if it is monotone, either increasing or decreasing. The result of interaction of predicates with crossover, defines their stability. Stable predicates are unchanged by crossover, semi-stable predicates are unlikely to be disrupted by crossover (degrade gracefully), while unstable predicates are disrupted by crossover. These concepts enable a more generalised explanation for schemata interaction. Global predicates are only influenced by the action of crossover, whilst local predicates are influenced by reproduction and crossover. Vose proposed the following conjecture regarding the classification of monotone predicates:

"The genetic paradigm will succeed when monotone increasing predicates are stable or semi-stable."

[Vose, 1991 p. 390]

1.4.4 GA And Constraints

1.4.4.1 Introduction

- ° **Definition 1.18 (Epistasis)** Epistasis is the differential in string fitness due to one string value being dependent on the presence or absence of other string values.

The work of John Holland can be applied successfully to low degree epistatic problems [Davis, 1985]. (We mentioned earlier how Holland recognised that non-linearity or false peak was an

obstacle to adaptation.) However, epistatic problems which involve constraints require modifying the simple GA approach. The basic architecture of GAs does not include their applicability to constrained search, i.e. *it is not well matched to it* [Goldberg, 1987] because there is no provision for satisfying constraints. Attempts to apply GAs to constrained optimisation problems use two main concepts:

- (1) Penalising strings which violate constraints.
- (2) Modifying GA operators to ensure constraints are satisfied.

The first concept combines the original objective function and a penalty function which assigns a constraint violation cost. The second concept includes using modified GA operators to ensure only feasible solutions are produced by repairing strings and hiding constraints in special representations which use specialised operators.

Other techniques involve using the cost objective and penalty objective as vector elements (e.g. Vector Evaluated Genetic Algorithm or VEGA [Schaffer, 1984]) or direct Pareto techniques [Goldberg, 1989] for allocating copies to individuals during reproduction. Both techniques eliminate the need to combine the cost and penalty objective, as in paradigm (1). VEGA uses multi-criteria functions where each criteria in the evaluation function are represented by equal sized sub-populations. Reproductive selection is carried out separately on each population, whilst crossover was performed across sub-population boundaries. One drawback with this approach however, was that selecting from the best in each sub-population created the potential for bias against individuals which were better across several criteria. The performance of VEGA was poor on multi-objective parameter tuning problems, where several criteria need to be optimised but cannot be reduced to a single objective. The Pareto optimality approach to multi-objective or vector valued optimisation attempts to produce a single P -set of points, where each point is a group of parameter settings. Each point in the P -set has a better objective function value for each parameter (non-dominated) than other points for the same parameters (dominated). The Pareto formulation of Goldberg [1989] obtains very good results on large feasible set covering problems but further work found difficulties.

1.4.4.2 Penalty Function

The penalty function method adjusts the fitness of a string in relation to any violated constraints. A constrained optimisation problem is transformed into an unconstrained one by associating a cost or penalty with constraint violations. In a constrained optimisation problem the goal is to minimise an evaluation $e(p)$ at a point p such that p is in the set F of acceptable points, (F is defined in terms of the constraints to be satisfied) [Richardson et al., 1989].

We can demonstrate the penalty function method using a *set covering problem* (SCP). In an SCP, there is an $v \times l$ matrix where $a_{ji} \in \{0, 1\}$, and each column has a cost variable c_1, c_2, \dots, c_l . The objective of SCP is to find the lowest total cost for a set S of columns, in which every row has at least one binary digit equal to 1, given as

$$\text{minimise } \sum_{i \in S} c_i \quad \text{subject to } \exists j \ a_{ji} = 1 \text{ for } j = 1, 2, \dots, v. \quad (1.7)$$

The following example SCP can be represented as an l length binary string S_{bin} in which if $S_{\text{bin}}i = 1$ then column $i \in S$, or $i \notin S$.

l	1	2	3
c_i	5	2	7
	1	1	1
	0	1	0
	1	0	0

A GA evaluation function $e(S_{\text{bin}})$ calculates the total cost for columns in S and adds a penalty function Φ cost associated with the number of constraint violations (uncovered rows), as follows

$$\text{minimise } e(S_{\text{bin}}) = \sum_{i=1}^l S_{\text{bin}}i \cdot c_i + \Phi(S_{\text{bin}}) \quad \text{subject to } \exists j \ a_{ji} = 1 \text{ for } j = 1, 2, \dots, v \quad (1.8)$$

For example, Richardson *et al.* [1989] used a penalty function in evaluating a binary string representation for the SCP as $\Phi = l \cdot (\text{number of uncovered rows})^2$. So for our example SCP an $l = 3$ length binary string $\{1,0,1\}$ has a fitness $e(1,0,1) = 12 + 3 = 15$. The GA searches the binary space for optimal solutions which satisfy the constraints (no uncovered rows

e.g. $\Phi = 0$) but at the same time process binary strings which do not satisfy the constraints (e.g. $\Phi > 0$).

Some researchers believe that penalty functions should be harsh, but all the population could contribute to the success of the GA. For example, if the optimal solution is 110, changing the 1's to 0's give infeasible solutions 010 and 100 which are only separated from the optimal by a hamming distance of one. If these near neighbours are heavily penalised the optimal solution may be missed. Therefore an evaluation function must preserve information which balances with the pressure for feasibility. One approach taken to construct a penalty function involves calculating the increase in fitness function cost in turning an infeasible solution into a feasible one, (basis of heuristic best first strategy). In the SCP this penalty function calculates the increased cost in adding columns to the set S of columns represented by the string S_{bin} until the constraints are satisfied. Research results from Richardson *et al.* [1989] using harsh, soft and softer penalty functions suggest more accurate estimates of the penalty function costs involved in making constraint violated solutions satisfy the constraints, make for better penalties. They found the softer penalty function made no distinction between feasible and infeasible solutions, where the search wandered aimlessly or suffered premature convergence. There is a limit to the effective softness used in the penalty functions, when the penalty for each constraint frequently falls below the expected cost of completion. From their work they concluded that penalties which are functions of the distance from feasibility perform better than those which are functions of the number of violated constraints, and that penalties should be as accurate as possible. These results highlight the difficulty in deciding how to quantify constraint violations which will allow the GA sufficient pressure to maintain feasible strings yet prevent premature convergence.

The approach taken by Siedlecki and Sklansky [1989] is to analyse the position of the population before reproduction, and balance a penalty function with the value of the objective function to achieve a specific distribution of the new population in the search space. The GA is encouraged to search for isolated parts of the feasible region but not end up wandering in the infeasible parts. Control of the GA is managed by changing the value of a positive constant penalty coefficient α , shown in the following evaluation function

$$\text{minimise } e(S_{\text{bin}}) = \sum_{i=1}^l S_{\text{bin}}^i \cdot c_i + \alpha \Phi(S_{\text{bin}}) \quad \text{subject to } \exists j a_{ji} = 1 \text{ for } j = 1, 2, \dots, v \quad (1.9)$$

This control of the new population distribution can take place in the reproduction stage without compromising the principles of the GA. The population distribution depends upon the value of the penalty coefficient and consequently on the balance between an original optimisation criterion and the penalty function. Therefore, by changing the value of the penalty coefficient the shape of the distribution of parents to be selected for crossover can be adjusted. Using a penalty coefficient allows the penalty function information to be used to guide the GA through the infeasible parts of the search space. However, the approach may only be useful when optimal solutions occur inside the feasible region and not on the boundary.

1.4.4.3 Specialist Crossover Operators

The location of crossover events may be sensitive to the contents of the chromosome [Alberts et al., 1983]. The representation and the incorporation of heuristics into the crossover operator are highly correlated. Grefenstette *et al.* [1985] showed that merely preserving the order of string values results in poorly performing GAs. It may be necessary to incorporate heuristics (problem specific knowledge) into the GA in order to make it competitive. We need to be aware of the established precepts in the design of representations which support the use of binary alphabets.

- ° **Definition 1.19** (GA-Hard) A GA-hard problem is both epistatic and misleading. [Bethke, 1981]
- ° **Definition 1.20** (*o*-schema) The *o*-schema defines a subset of schemata which have the same string values at the same string positions, where the values of unspecified positions are ignored (i.e. "#").

A number of different crossover operators have been developed to work with real-coded representations. These crossover operators are faced with the problem of constructing feasible offspring and yet enable good building blocks to be inherited from the parents. Grefenstette *et al.* [1985] suggest that the TSP is GA-Hard and may not be suitable for the GA. Attempts to design crossover operators for real-coded representations to solve the TSP have demonstrated

the coding difficulties faced by GA designers. The *alternating edges* crossover operator [Grefenstette et al., 1985] developed for an *adjacency representation* to tackle the TSP, has poor results because good sub-tours are disrupted. The *adjacency representation* describes a tour by a list of cities, where there is an edge in the tour from city i to city j only if the allele in position i is j . An example five city TSP can be represented as

Path Tour	Adjacency Tour
(1 3 5 4 2)	(3 1 5 2 4) city j 1 2 3 4 5 position i
	3 city j
position i	
1	3
3	5
5	4
4	2
2	1

Any tour has exactly one adjacency list representation but does not allow the classical crossover operation. The *alternating edges* crossover operator developed for the adjacency representation starts by choosing a random edge from one parent, then extends the partial tour by choosing the appropriate edge from the other parent, and continue alternating parents. If selecting an edge would introduce a cycle (repeated city) then select a random edge.

sequence	1	5	3		2	6x	4							
parent 1	(2	3	4	5	6	1)	parent 2 (2	5	1	6	4	3)	city j	
	1	2	3	4	5	6		1	2	3	4	5	6	position i
offspring	(2	5	4	1	6	3)	city j							
	1	2	3	4	5	6	position i							

The crossover operator starts with parent 1 edge (1 2). When the sequence reaches 6 the selection is invalid, so a new edge is created (4 1). The results for this operator are poor because good sub-tours are disrupted.

The *cycle* crossover operator (CX) suggested by Oliver *et al.* [1987] uses common subsets of cities between parents and exchanges them. This operator performs poorly which suggests that retaining relative city positions is not as important as the links between cities. Whitley [1989] supports this view and states that the power of recombination for the TSP, is to focus on the exchange of edges (city to city links). The *edge recombination* operator [Whitley et al.,

1989] retains up to 95% - 99% of parent information and can be shown to be supported by the Schemata Theorem, (i.e. the edge recombination operator is manipulating schemata). The edge recombination operator is useful in scheduling problems because it does not use edge cost information, only string performance. Results for the MPX crossover which uses the idea of a donating and receiving parent where the relative order of the receiving parent remains [Goldberg and Lingle, 1985], shows that with a high probability, low order *o*-schemata survive to lead to optimal or near optimal results because it searches among both orderings and string value combinations that lead to good fitness. The advantages of the MPX operator is that the relative city positions are retained and implicit mutations are introduced.

Heuristic GAs incorporate problem specific knowledge in the crossover operator. The combination of hill climbing with GA depends upon the availability of good starting points. The Grefenstette *heuristic* crossover "glues" good edges together but cannot fine tune solutions. City to city links are selected from the parents by choosing the shortest edge, or randomly if a cycle is introduced. The technique used to help towards fine-tuning solutions is the Lin and Kernighan [1965] 2-opt operator. The Grefenstette *et al.* [1985] *subtour-chunking* crossover performance was better than the *alternating edges* operator but still not very good. In the subtour-chunking crossover operator a random length subtour is selected, extended by random length subtours from alternating parents. If an edge creates cycling choose a random edge. The *analogous* crossover operator [Davidor, 1991] matches parameters for crossover based upon the string value characteristics rather than the string values. For example, crossing parent strings which represent driving routes will use shared stretches of road (e.g. edges (8 4) and (10 3)) to align them for crossover.

parent 1 tour -	1	2	8	4	6	9	10	3	5	7
parent 2 tour -	5	1	8	4	7	2	10	3	6	9
offspring tour -	1	2	8	4	7	2	10	3	5	7

Disruption in the offspring caused by analogous crossover operator can be corrected by random selection of repeated (e.g. 2 and 7) and unrepresented cities (e.g. 6 and 9). The analogous cross site preserves the "orderliness" of strings, and has an intuitive justification and biological motivation. Where multi-bit parameters are used, segregation crossing between features becomes

an issue. This may have a disruptive effect on the string values and consequently the string characteristics. [Smith, 1980; Greene and Smith, 1987; Shaefer, 1987]

A number of crossover operators have been developed which use a template/mask to control offspring generation. The punctuated crossover operator [Schaffer and Morishima, 1987] uses a representation where crossover points are recorded by the punctuation symbol. The corresponding crossover will swap allele transfers from one parent to the other depending upon occurrence of the symbol. The string representation is in two sections (double the normal length). The first records the corresponding crossover points, 1 = yes, 0 = no. The loci in the second section are the objective function values. The mutation operator is allowed access to both sections of the string. For example

before crossover

parent 1 - a a a a a a a! b b b b b b b

parent 2 - c c c c! d d d d d d! e e e e

during crossover

offspring 1 - a a a a
 a a a a d d d
 a a a a d d d b b b
 a a a a d d d b b b e e e e

after crossover

offspring 1 - a a a a d d d b b b e e e e

offspring 2 - c c c c! a a a! d d d! b b b b

Syswerda's [1989] uniform crossover used a template to decide which child receives the parent bits. Davis' [1991] uniform order-based template crossover is used to decide which child receives the parent bits. The Davis uniform crossover operator is a masking string which determines the action of the crossover mechanism. Using the mask from left to right, bit 0 tells the crossover operator to select the bit value from parent 1, bit 1 tells the crossover operator to select from parent 2. The second child is created by switching the meaning of the binary mask values. This action of the crossover operator is to allow the parents to exchange values with a uniform distribution. These masks can allow the action of the traditional one or two-point crossover

and can link together string values which are separated along the string [Syswerda, 1989]. Syswerda suggests his results show that uniform crossover combines schemata more effectively than one or two-point crossover.

1.4.4.4 Specialist Mutation Operators

Different mutation operators had to be devised to cope with real-coded alphabets. Davies' adjacency mutation operator used in the study using real-coded feature values can *creep* around the search space. The *scramble-sublist* mutation operator takes a sublist of the chromosome, changes the order and places them back into the parent. A variation on this operator is used where there is a parameter of the sublist length. It is suggested [Grefenstette, 1987] that local search may be useful as a mutation operator, (prevented from being trapped in local minima by the GA operators).

1.4.4.5 Inversion Operators

The *inversion* operator has been studied [Frantz, 1972] as a method for linking together separated string values, by randomly selecting a point to cut a string and then inverting one of the string sections as in,

before inversion -	5	1	8	4	7		10	3	6	9
after inversion -	5	1	8	4	7		9	6	3	10

Although inversion attempts to reduce the error rate for schemata disruption and increase the combination rate of the crossover operator, masking allows the bits to remain separated and therefore there is no need to use inversion [Syswerda, 1989]. If a population contains building blocks where the string values are not tightly linked together, the inversion operator will increase the probability of reconstructing good building blocks. Inversion may be useful in limited cases but in general GAs need the power of recombination. Rosenberg [1967] used linkage factors associated with each string value to help provide tight linkage by selecting a crossover site according to a probability distribution over the linkage factors (i.e. selecting a cross site where the values are different). Frantz [1972] used a simple genetic algorithm to test the linkage effects on several functions where the linkage was disrupted (chromosome ordering), finding that there is a correlation between tight linkage and rate of improvement. The theory was not fully

confirmed, due to the insufficiently difficult functions used in the tests (e.g. weak non-linearities, and short strings). Bethke [1981] in his thesis characterised the set of functions which are genetically optimisable in terms of the Walsh transforms of the function, and showed that the coefficients involved can be estimated during the search to determine whether the function can be optimised genetically.

1.5 Summary

Constraint satisfaction problems (CSPs) can be found in many application areas of artificial intelligence (AI) and research is motivated by the need to provide efficient methods to tackle them. In general, CSPs are *NP*-hard and techniques used to tackle them suffer from combinatorial explosion preventing traditional search methods from being useful, due to the time taken to find solutions. For CSPs with optimisation functions (CSOPs) heuristics giving an accurate estimation of the cost function could be used to help find solutions, but would be unlikely to be able to solve very large problems. Also heuristics are domain specific, and good heuristics are sometimes difficult or expensive to find. Partial constraint satisfaction problems (PCSP) have constraints that cannot always be completely satisfied. Our objective in this research is to design and empirically test a generic search strategy based on stochastic search which can tackle a class of PCSPs, which includes solvable and unsolvable CSOPs. We believe a successful approach can be achieved by the combination of a robust genetic algorithm (GA) with local improvement which will provide a strategy with the ability to efficiently exploit PCSPs. In order to exploit the features of PCSPs, this new GA strategy called GAcSP has been designed which incorporates a template type crossover and hill-climbing component (HC). GAcSP combines the robust global power of the GA with the specialist power of the HC. We hope to show that the HC component can exploit domain specific knowledge without compromising the GA generic search performance. The performance of GAcSP is demonstrated by tackling two *NP*-hard problems, namely the processors configuration problem and the car sequencing problem.

GAs were originally developed by John Holland [1975] as artificial systems based on the adaptive processes of natural systems. With certain search space conditions and the GA evolving a population of binary strings by simple syntactic actions upon them, it could converge on solutions

which were at or close to the global optimum. The canonical or simple GA has three main operators: reproduction, crossover and mutation. This simple model of GA assumes that binary strings are fixed length, a finite dynamic population of strings represents what is known about the search space and each string has a relative ability (fitness) to survive and produce offspring. The GA starts by generating a random population of binary strings, then for each cycle reproduction, crossover and mutation act upon this population of strings. Reproduction generates a matepool of potential parents by selecting strings from the population with a bias towards fitter strings (low fitness for minimisation problems). The one-point crossover operator randomly selects two parent strings from the matepool, cutting and exchanging the parent strings to create two offspring. Mutation is used with a low probability to change an offspring string element. The process of selecting parents and creating offspring continues until a new population is generated.

The *Schemata Theorem* of GAs allows us to quantify the survival probability of strings and predict the behaviour of the SGA. The Schemata Theorem is based on the idea that the GA processes string similarities (templates), called schemata (singular schema H) defined over the alphabet $\{0, 1, \#\}$, where the metasymbol $\#$ can match 0 or 1. According to the Schemata Theorem the power of the GA can be demonstrated by counting the number of schemata in processing a population of strings. Holland's $O(n^3)$ schemata processing estimate, means that through the processing of only n string structures each generation, a GA processes something like n^3 schemata. The growth of schemata depends upon the ratio of schema fitness, calculated as the average fitness of the set of strings it describes, to the population average fitness. Above average schemata will generate an exponentially increasing number of copies with those below average decreasing. The survival of schemata also depend upon the disruptive effects of crossover and mutation. The probability that crossover will disrupt schemata depends upon how far apart string values are located and with mutation the number of values. We can use the Schemata Theorem to predict that short high fitness schemata called *building blocks* will increase exponentially, from one generation to the next. If the short schemata accurately reflect the optimisation of the search space, the GA should converge to an optimal or near-optimal solution.

The Schemata Theorem and the canonical GA supports the use of binary string representations and was accepted early on as the best way to maximise the processing power of the GA. There are a number of reasons why binary representations have dominated GA research including the simple analysis of binary vectors, the elegance of GA operators, and the requirements of computational speed. Furthermore, the Schemata Theorem suggests that small alphabets maximise the number of schemata available for GA processing. However, empirical evidence supports the use of real-coded representations but does not attempt to incorporate Holland's Schemata Theory. Many problems cannot be expressed in terms of a binary alphabet and requires more sophisticated data structures and GA operator modifications. Another reason for using real-coded alphabets is the avoidance of *hamming cliffs* and other artifacts of mutation where binary coded GAs can be stopped from reaching points in the search space. Although real-coded alphabets can converge more quickly than small coded ones, the quality of the solution can degrade with increasing alphabet cardinality. Real-coded alphabets can also reduce the combinatorial dimension of the problem reducing the opportunity for GA deception. One important problem in extending the Schemata Theorem to real-coded and position dependent representations is the "type" of building block available. The analysis of representation space can be developed by investigating building blocks that compose a schema by looking at the relationships between the problem parameters. GA success in finding optimal solutions depends upon these schemata correlating with performance. The Walsh function analysis devised by Bethke [1981] and extended by Holland [Goldberg, 1989] to include non-uniform populations can be used to measure this correlation for binary representations. Mason [1991] has extended the Walsh function analysis for real-coded alphabets and Goldberg has presented the notion of *virtual alphabets* to support a theory of operators for real-coded GAs.

Supporting real-coded representations, Antonisse [1989] questioned the analytic framework used to justify minimal alphabets. In particular critical issues concerned with how the symbolic encoding of the search space effects the representational power of the search, and what legal structures can be encoded. Antonisse reinterprets the use of the "#" symbol in analysing schemata where the class of strings is a single subset extending it into the case where every possible subset of individuals is counted as a schema. In Antonisse's interpretation, the counting argument to

express all schemata is not $k + 1$ but $2^{(k-1)}$ and therefore, the more expressive real-coded alphabet is seen to carry much more power, providing *finer-grained* tools for the construction of adaptive plans. However, there are limitations with schemata counting arguments and Schemata Theorem analysis. Vose [1991] developed an alternative Schemata Theorem framework for measuring the effectiveness of crossover operators and disruption rates of *generalised* schemata, which they call *predicates*. This alternative interpretation of the Schemata Theorem emphasises the importance of schemata disruption on what building blocks can be formed from the crossover schemata interaction. Schemata can be regarded as predicates which map binary strings into the set {true, false} and allows a more generalised view of schemata interaction.

Although the GA can be applied successfully to low degree epistatic problems, problems which involve constraints require modifying the simple GA approach. The basic GA architecture has no provision for satisfying constraints. Attempts to apply GAs to constrained optimisation problems use two main concepts penalising strings which violate constraints and modifying GA operators to ensure constraints are satisfied. The penalty function method adjusts the fitness of a string in relation to any violated constraints, thereby a constrained optimisation problem is transformed into an unconstrained one by associating a cost or penalty with constraint violations. An evaluation function must preserve information which balances with the pressure for feasibility to avoid penalising strings close to the optimal solution. One approach taken to construct a penalty function involves calculating the increase in fitness function cost in turning an infeasible solution into a feasible one. Research results using harsh, soft and softer penalty functions suggest more accurate estimates of the penalty function costs involved in making constraint violated solutions satisfy the constraints, make for better penalties.

The representation and the incorporation of heuristics into the crossover operator are highly correlated and show that merely preserving the order of string values results in poorly performing GAs. It may be necessary to incorporate heuristics (problem specific knowledge) into the GA in order to make it competitive. Different crossover operators have been developed to work with real-coded representations and are faced with the problem of constructing feasible offspring and yet enable parental building blocks to be inherited. A number of crossover operators have been developed which use a template/mask to control offspring generation. For example, Syswerda's

uniform crossover and Davis' order-based crossover uses a template to decide which child receives the parent bits. The action of the crossover operator allows the parents to exchange values with a uniform distribution. They can also replicate the action of the traditional one or two-point crossover and link together separated string values.

Different operators had to be devised to cope with real-coded alphabets. The *inversion* operator is a method for linking together separated string values, by randomly selecting a point to cut a string and then inverting one of the string sections. Inversion attempts to reduce the error rate for schemata disruption and increase the combination rate of the crossover operator by increasing the probability of reconstructing good building blocks. However, results in using inversion suggests that GAs need the power of recombination

Chapter 2 Design Of A New GA For Solving PCSPs

2.1 Analysis Of Applying GA To PCSP

2.1.1 Control Strategy

We have developed a control strategy for tackling PCSPs, based upon a standard GA [Grefenstette, 1984] and local improvement techniques. In order to analyse the task of applying GAs to PCSPs we need to look at the essential GA components and PCSP features in detail. If we can identify and understand these PCSP features it may be possible to exploit them in developing the GA strategy. We have already outlined important PCSP features in Section 1.3.1.2. In this section we shall try to understand how they have influenced the design of the GAcSP. The following issues refer to GA design and PCSP features:

- (1) Elements of the PCSP (i.e. variables and values) lend themselves to a natural GA string representation.
- (2) Constraints can be used to prune the search space by making parts of the search space infeasible.
- (3) Constraints in one part of the solution are propagated to other parts by the repair and hill-climbing technique.
- (4) Using a GA to tackle a class of problems which has a well understood set of assumptions, enables certain predictions about the behaviour of the algorithm to be made.
- (5) Ordering of PCSP variables and values can enhance or hinder PCSP search techniques, but the GA need not rely upon any assumptions regarding ordering.
- (6) The PCSP constraint relationships can assist GA search, providing a guide through the search space towards optimal solutions.
- (7) GAs tackling PCSPs have the opportunity to adapt to a structured search space characterised by the relationship between variables and values.

(8) GAs have the ability to record constraint interactions in the finite string population.

Constraint interactions can have a significant impact on the design of GA components. Although we will examine GA components, it must be emphasised that GAs are a heuristic *system*. By system, we mean that the interacting components forming the GA and the search space characteristics are inter-dependent. The search space characteristics are defined by the representation and objective function, and will determine the design of GA operators. PCSP constraints will influence the dynamics of the population and GA operators. Because GAs manipulate a population of representation structures, it is important to understand the dynamics of this population. GA operators need to be designed which can improve the population string structures towards optimality and not severely restrict the global ability of the search. An additional GA component in our strategy is the use of an optional hill-climber.

In Sections 2.1.2 and 2.1.3 we look at the impact on the search space of representation design and constraints, GA crossover operator design (Section 2.1.4), mutation design analysis (Section 2.1.5), and population dynamics (Section 2.1.6).

2.1.2 Representation

The representation and the GA operators need to work together in a synergistic way. The GA crossover operator manipulates chromosome-like structures in a way its natural counterpart does. String representations are suitable because of their structural similarity to chromosomes.

The "art" of successfully applying GAs to a particular problem involves deciding the most suitable way of representing the problem. What help is available within the field of GAs in making this decision ? Help can come from a number of different sources. Techniques which may help, include the following quantitative and qualitative methods.

- Quantitative methods for analysing binary-coded representations. These include Walsh function analysis, and dynamic epistatic analysis.
- Research using real-coded representations.
- Goldberg's principles which should be adopted by GA developers - the *principle of meaningful*

building blocks and the *principle of minimal alphabets* (See Section 1.4.3.1).

- Qualitative analysis using Davidor's *base* and *parasitic* epistasis, [Davidor, 1991].
- Techniques currently employed in tackling the problem.

The first and perhaps most critical decision facing representation designers is whether to use a binary or real-coded representation. Surprisingly, this choice may depend more on familiarity or preference, than suitability. Which techniques can assist in the development of a suitable representation will depend upon this initial choice. Binary-coded representations are supported by established GA theory and operators; real-coded representations can be more appealing but require more careful analysis. We first look at an example binary representation and then consider a real-coded case where we find it is possible to combine the two. This is justified by an analysis of the crossover operator in Section 2.1.4.1.

We need to consider the essential features of the PCSP in designing an appropriate representation. The formal definition presented in Section 1.3.1.1 gives us the framework upon which to design the binary representation. Section 2.1.2.1 outlines an example binary representation.

2.1.2.1 Binary Representation

Based upon the formal PCSP Definition 1.2 we can design a simple binary representation as follows:

In general, a string S_{bin} is composed of N binary integers, $S_{\text{bin}} = (b_1, b_2, \dots, b_N)$.

Without loss of generality, each binary integer b_i is mapped from the value v_{ij} taken from the PCSP variable domain v_i . Formally the mapping f is from the integer into the binary,

$$f: \text{int} \rightarrow \text{binary string}.$$

An example binary string is shown with the corresponding PCSP values:

string position - PCSP variable	1	2	3	4	5	6	7	8	9	10
solution tuple - binary coded	001	010	011	100	010	101	100	001	011	101
equivalent - PCSP value	1	2	3	4	2	5	4	1	3	5

The length l of S_{bin} will depend upon the number of variables N in the PCSP, and the greatest domain values for each $\max v_i$. To calculate the length l of the binary string, we need to first find the greatest value in each variable domain.

Note: If all the domains of a PCSP are equal (i.e. have the same elements),

$$\max v_1 = \max v_2 \dots = \max v_N.$$

Then map the integer $\max v_i$ into the binary $l = (\log_2(\max v_i) \cdot N)$ binary digits. For our example string above, $l = (3 \cdot 10) = 30$ binary digits.

The main perceived advantages of using a binary-coded representation are the opportunity to use traditional GA operators and support from the Schemata Theorem. Issues of using a traditional crossover concern whether the point of crossover should occur only between the binary integers (inter-chromosomal) or between binary digits (intra-chromosomal). This issue is related to the one discussed below regarding the decoding of the binary string by the objective function. Drawbacks to using binary-coded representations include string length, (i.e. decoding and encoding problems), problems such as hamming cliffs and slower convergence rates [Goldberg, 1989]. The decoding and encoding by the objective function with a binary string will increase the computational work load of the GA, especially with large populations. Another encoding and decoding problem depends upon the precision of the inverse mapping of the binary integers into the real values (i.e. PCSP values). This mapping can give rise to imprecision and can cause redundant real values. For example, if $\max v_1 = 6 \equiv 110$ then the binary integer $111 \equiv 7$ which could be formed would be a redundant value. Redundancy also occurs because binary mapping assumes an arithmetic progression (*common difference* of 1) of the values in v_i . As we shall soon see, we do not have to completely abandon the binary string representation and some of its advantages when we consider a real-coded representation presented in the next section.

2.1.2.2 Real-Coded Representation

Instead of using a binary-coded representation we can use a non-binary or *real-coded* representation. This seems more of a "natural" representation as suggested by Goldberg's second principle and the framework of the PCSP model. It results in a more compact string, and we can work directly with the PCSP domain values. So, instead of mapping the PCSP value to a binary integer we use the PCSP value directly, as in:

string position - PCSP variable	1	2	3	4	5	6	7	8	9	10
solution tuple - real coded	1	2	3	4	2	5	4	1	3	5

In general, a string S is composed of $l = N$ PCSP variable domain values, $S = (v_1, v_2, \dots, v_l)$.

We can consider the differences in approach between the GA and established PCSP solution techniques. The first important point is that the PCSP representation search space is a network of solution tuples (see Definition 1.3). Many PCSP solving techniques use a hierarchical search space or search tree. These PCSP techniques construct solution tuples by searching through the search space or search tree. More recent approaches attempt to find PCSP solutions by repairing solution tuples. Heuristic repair is one such method which has shown some success using a repair technique [Minton et al., 1991]. This distinction between constructing solution tuples or working directly with solution tuples is an important part of the GA approach. This gives the GA more freedom to move around the search space by providing immediate complete information from any solution tuple changes, so long as the tuple can be meaningfully interpreted. Furthermore, changing a single string value or values can be done with little computational effort, yet the effect of even such a simple change to the string can be significant for two reasons because:

- (1) The string fitness may depend upon the interaction between the changed value and other values in the string.
- (2) Constraints may become violated by the new value.

We can extend this idea by changing several values simultaneously or a group of contiguous

values. Improving the fitness of a group of string values can increase the string fitness. The level of increase in string fitness can depend upon the degree of interaction between string values. Also, changing a group of values could increase the possibility of violating constraints and making the string infeasible. The basis of the GA approach is to combine fitter groups of values from different solutions (parents) to create fitter strings (offspring). The GA reproduction, crossover and mutation operators are designed to explore and exploit these groups of values. String elements which depend upon other string elements due to constraints between them, can create dependency groups within the string. It is important to understand the effect of constraints on the GA search (i.e. points (1) and (2) above) in order to control the search process.

2.1.3 Exploiting Constraints

Constraints are an essential feature of the PCSP and play an important role in the development of the GA strategy. They influence the design of the GA operators by their impact upon the representation search space. Constraints handled by the objective function determine the shape of the search space. We have already discussed the building block approach of GAs and mentioned the interaction between individual string elements which leads to the building block hypothesis (see Section 1.4.2). This phenomena of interaction is termed *epistasis* and is an important concept in GAs. Epistasis is clearly visible in natural systems where certain physical characteristics result, and in artificial systems through the fitness function.

Constraints represent interacting relationships between PCSP variables and this can give rise to the phenomena of epistasis in two separate ways:

(1) The effect of epistasis in the GA is seen through the fitness function. However, not all strings generated by the GA are solution tuples to the problems tackled. For example, the valency of processors in the PCP and the production requirement in the CarSP are constraints that must be satisfied. These *hard* constraints have a direct effect on the structural search space.

(2) *Soft* constraints are the user requirements or the constraints of the problem which can be optimised. For example, minimising the mean internode distance in the PCP and ensuring

workstation capacity are satisfied in the CarSP. Soft constraints can be incorporated into the objective function which influence the search through the fitness function.

In order to analyse the effect of constraints upon the GA strategy we need to distinguish between these hard and soft constraints.

2.1.3.1 Hard Constraints

Hard constraints, are those that must be satisfied for the solution to be acceptable. These hard constraints can occur as fixed requirements (e.g. *production requirements*) and can be problem dependent. Hard constraints can be handled by a GA in a number of ways. Constraints can be hidden by a specially designed representation and set of operators, but this may restrict which type of constraints can be handled by our GA strategy. Hard constraints incorporated into the objective function using a *penalty function* approach require specific theoretical and empirical procedures for the combination. With penalty functions the search space becomes the space of all possible solution states, rather than solution states where the hard constraints are satisfied. Although research carried out by Richardson *et al.* [1989] has suggested useful principles in combining penalty functions with objective functions there are still no general reliable techniques available. An important drawback with using penalty functions is due to disadvantages in allowing the GA into infeasible parts of the search space, where solution states do not satisfy the hard constraints. In these cases, the measure of penalty cost which should be associated with constraint violation is difficult to accurately estimate. With high cost penalties the GA may find it is evaluating more infeasible strings than feasible, and could converge on the first feasible string found. Low cost penalties will not put sufficient pressure on the GA to generate and keep, feasible strings.

We need to consider the merits of these alternative techniques to hard-constraint handling in the light of our research objective which is to design a robust generic GA strategy for tackling PCSPs. The approach taken by the GAcSP is that any strings created by GAcSP in the initialisation or crossover operators are *repaired*, ensuring that the hard constraints are satisfied. This approach has the advantage that it can prune some of the search space by preventing the GA from entering infeasible areas of the search space and eliminating the difficulties

associated with estimating violation costs. If we use a GA strategy to tackle PCSPs which are tightly constrained the majority of the search space could consist of infeasible solutions. We can make the GA strategy more robust by allowing for the possibility of incorporating different hard constraints without needing to change the general strategy. We therefore adopt the approach of repairing each string created by the GA.

2.1.3.2 Soft Constraints

The objective function is designed to try and capture the soft constraints of the problem. These constraints are therefore directly optimised by the GA. Soft constraints can include resources, customer requirements, economic objectives or a combination of these and other costs. Combining different costs into a single objective function does not necessarily create the difficulties of combining an objective function with a hard constraint violation cost, as with the penalty function approach. This is because the penalty function approach involves combining two different objective measures into a single value. Also, unlike hard constraints, soft constraints need not necessarily be completely satisfied for a solution to be acceptable. The GA approach is to find optimal or near optimal solutions to PCSPs which maximise or minimise the soft-constraint violation measured through the fitness function.

One difficulty which these constraint interactions pose for the GA is in the design of appropriate operators - most notably the crossover operator, which is the *engine* of the GAcSP strategy.

2.1.4 Crossover Design Analysis

2.1.4.1 Crossover Operator

Whether binary or real-coded representations are used, the crossover operator must have the ability to propagate building blocks throughout the population and to create new ones. How can we ensure that the action of a crossover operator on a real-coded representation will behave as predicted by the Schemata Theorem ? Goldberg's *principle of meaningful building blocks* suggests that it is essential for building blocks to be a tightly linked group of string elements when a one-point crossover is used. The issue of constraints however provides a potential difficulty, in that string elements which interact could do so over any distance along

the string. Thus the building blocks will not necessarily be short but may extend to some distance along the string. This is a serious issue which must be addressed because the power of the search (i.e. $O(n^3)$ [Holland, 1975]) depends upon the survival and propagation of building blocks.

The Schemata Theorem predicts that with one-point crossover these groups of tightly linked string elements will have a chance of survival depending upon their distance from each other and the string length. Alternative crossover operators and the inversion operator do not offer adequate answers to these difficulties. Clearly, what is needed is a crossover operator which will allow the formation of building blocks which are not short but have elements distributed along the string. One-point and two-point crossover preserve good schemata only if they are tightly packed. If we extend this idea and consider a multi-point crossover we can be certain that the outcome will be schema disruption to the point of becoming a pure random search [De Jong, 1975]. The *uniform crossover* developed by Syswerda [1991] is a multi-point crossover operator with the ability to identify building blocks by remembering crossover points. Although it has been shown that the uniform crossover may devastate tightly packed schemata it can perform better on schemata which are not [Davis, 1991]. With this distributed ability of the uniform crossover, we adapted it to our requirements, and called it the *uniform adaptive crossover* (UAX).

- ° **Definition 2.1** (*On-line*) *On-line* performance is the mean of all trials. Formally where $f_e(i)$ is the average i th function value and T the total number of evaluations

$$\text{On-line performance } x_e(T) = 1/T \cdot \sum_{i=1}^T f_e(i).$$

- ° **Definition 2.2** (*Off-line*) *Off-line* performance is the mean of the best previous trials $f_e^*(i)$ at each time (or trial) T .

$$\text{Off-line performance } x_e(T) = 1/T \cdot \sum_{i=1}^T f_e^*(i).$$

The UAX operator will allow us to cut the string into sections along the string length. Research carried out into increasing the number of cut-points used in the crossover operator has shown

that the performance of the GA can be degraded until the resulting search is nothing more than a *random walk*. (The *on-line* and *off-line* performance measures are degraded [De Jong, 1975].) The reason for this loss in performance is due to schema survival. As the number of crossover points is increased the possibility of disrupting (i.e. cutting into) valuable schema will also increase. In addition, if we interpret David Goldberg's first principle where the epistatic interaction between building blocks is limited in the GA process, then our crossover needs to be designed such that this is not a problem. Therefore, at the same time as being able to capture sections of the string, we need a way to retain and propagate these new building blocks, preventing them from being destroyed.

The points at which UAX cuts the parent strings during recombination are determined by the parent binary templates. A binary template for each string consists of a set of binary values, where each binary value is associated with a particular string value. Each binary template is stored on the string along with the associated string values, as shown in the following example,

string position - PCSP variable	1	2	3	4	5	6	7	8	9	10
solution tuple - real coded	1	2	3	4	2	5	4	1	3	5
binary template	0	1	0	0	1	1	0	0	1	0

Every string in the population has a binary template; the initial population of strings have randomly generated binary templates and the UAX mechanism ensures that offspring will inherit the parent binary template values. The binary templates record successful cut-points and effectively creates an underlying binary space. Using the new UAX operator upon this underlying binary search space will allow the offspring the opportunity to inherit these building blocks. Through the manipulation of this underlying binary space, important real-coded building blocks will be inherited, which will lead the GA towards "optimal" solutions. An analysis of the population dynamics in respect of these building blocks will enable us to be more confident in this approach.

2.1.5 Mutation Design Analysis

The following two functions, repairer and hill-climber have the same search enhancement ability as the mutation operator, with additional advantages.

2.1.5.1 Repairer

The Repairer ensures the hard PCSP constraints are satisfied by changing individual string values. We have seen from Section 1.4.3.2 that making small changes can allow the GA to *creep* around the search space. The repair function can focus on making changes at the level of a single string element. This is an important ability for the GA to possess because the crossover operator can encourage building blocks but needs a finer tuned operator to increase diversity and aid improvement. The string repair function and hill-climber help accomplish this task.

2.1.5.2 Hill-Climber

Alphabets larger than binary can avoid binary coding problems such as hamming cliffs [Goldberg, 1990]. Although GAs tackling problems with large alphabet codings can converge more quickly in finding near optimal solutions the quality of solutions can degrade as the alphabet size increases [Goldberg, 1990]. The convergence rate of GAs is important factor in determining solution quality. One way to control the convergence rate of real-coded GAs is by population size - larger populations can sustain search through increased diversity. Another, is to increase the diversity of the population by introducing changes to offspring strings. This supports the need for a localised hill-climber in order to increase the potential quality of solutions and compensate for the possible loss of performance through the use of large alphabets.

The hill-climber also makes element improvements, but changes two elements instead of one as in the repair function. Diversity is maintained because PCSP values can be made to appear at any string position through the action of the hill-climber. The aim of both functions is to increase the string fitness by making small changes over the length of the string. Building blocks can be improved and these improvements will be propagated throughout the population. The action of the HC will put pressure on the GA to seek further improvements because the

population average fitness will increase as each string improves in fitness. Because the action of the repair and hill-climb functions is to directly alter the real-coded representation, binary problems such as hamming-cliffs normally associated with binary strings can be avoided.

2.1.6 Population Dynamics Analysis

We can describe the dynamics of the population in two ways - external and internal. The external dynamics of the population describes how the population of strings is manipulated by the GA and this will be covered by Section 2.2.8.1. The internal dynamics describes how the elements which make up the strings change the sampling nature of the population. The following analysis describes these internal dynamics. The internal properties of the population depend upon how an initial population of strings is manipulated by the GA.

Through the action of the reproduction and UAX operators the population of strings will develop binary patterns which will explore and exploit the real-coded search space. We use the principles of schemata processing to describe the action of the reproduction and UAX operators on the population of strings. For the purpose of this analysis a string can be regarded as having two parts - the first part consists of the real-coded values, and the second is the binary template. In following example string we define the binary template in terms of a schema (e.g. schema fitness $f_{00} = 5$) and show the corresponding real-coded values at the schema fixed positions.

string position -	1	2	3	4	5	6	7	8	9	10
string 1										
real-coded values			3						4	
schema - $f_{00} = 5$	#	#	0	#	#	#	#	#	0	#

Consider the following four strings,

string position -	1	2	3	4	5	6	7	8	9	10
string 1										
real-coded values			3						4	
schema 1 - $f_{00} = 5$	#	#	0	#	#	#	#	#	0	#

real-coded	values			2						3
schema	2 - f_{01} = 4	#	#	0	#	#	#	#	#	1 #

real-coded	values			5						4
schema	3 - f_{10} = 7	#	#	1	#	#	#	#	#	0 #

real-coded	values			4						2	
schema	4 - f ₁₁ = 3	#	#	1	#	#	#	#	#	1	#

The binary template schemata f_{00} , f_{01} , f_{10} and f_{11} are an underlying representation for real-coded values. As the GA progresses, strings with new combinations of real-coded values can be created by the UAX or mutation operators which may provide a better fitness for the schemata. If string 5 is created as

real-coded	values			4						3
schema	4 - f00 = 9	#	#	0	#	#	#	#	#	0

then schemata will compete with the new fitness values. These schemata competitions will continue until the schema associated with real-coded values of above average fitness, dominate the population. Furthermore, an increase in real-coded fitness will result in the population

average fitness being increased, causing the competing below average fitness schemata to have decreased opportunity to mate. The important implicit parallelism which gives the GA its power is effectively increased and there is an empowered implicit parallelism in the GA process. It is also conjectured that this increased competition can prolong the useful life of the GA, retarding convergence.

The limitation with most approaches to tackling PCSPs is in making changes to a single solution (e.g hill-climbing) or by following a single path through the search space. The limitation with both these approaches, is that they are serial and very localised. However, GAs manipulate a population of solutions and this enables them to simultaneously test different parts of the search space. This simultaneous testing gives the GA a parallel search ability. By exploiting search space information the GA can effectively explore different parts of the search space.

2.1.7 A GA System

We have looked at GA components in detail in the previous Sections (2.1.1 - 2.1.6) and analysed the impact of PCSP features on their design. This has been undertaken with our research goal clearly in mind - to develop a robust generic GA strategy to tackle PCSPs. The analysis of each GA component in detail has been considered in terms of its inter-dependence within the GA system. The components should work together in a synergistic way, allowing necessary flexibility where possible. The components outlined in this section form a GA system which we call GAcSP.

2.2 What Is GAcSP ?

2.2.1 Outline Of GAcSP

The heuristic strategy which we call GAcSP is distinguished from the standard or simple GA by the integration of hill-climbing, string repair, best fitness string selection and an adaptive template type crossover operator. As we have seen from the previous analysis, this combination is specialised for solving PCSPs and exploits the characteristics of PCSPs.

We hope to show that the combination of a GA and HC is synergistic, exploiting the abilities

of each method. Although the GA is a robust technique for finding near optimal solutions in combinatorial search spaces, it generally lacks a local improvement ability. This local ability is provided by combining the GA with a simple HC. The GAcSP uses a new uniform adaptive crossover (UAX) which generates offspring strings from parents strings to explore and test new areas of the search space. After crossover, the offspring string is repaired and hill-climbed. These repair and hill-climb functions act as a mutation operator altering individual string elements. The HC increases the fitness for every offspring after crossover generation, before the string is expected to compete with its peers.

There are five components of the GAcSP strategy, namely:

- PCSP representation
- Objective function
- GAcSP operators (initialisation, reproduction, and crossover)
- GAcSP parameters
- Population dynamics

The following Sections 2.2.2 - 2.2.8 will describe and discuss each of these components in detail.

2.2.2 PCSP Representation Framework

From the PCSP definition given in Section 1.3.1.1 and its solution tuples, we can define a *standard string* representation which is natural to the problem. Here each PCSP variable x_i in Z relates to a string position i in an l length string, and at that string position has an assigned PCSP value v_{ij} taken from its domain v_i in D .

$$Z = \{1, 2, \dots, l\},$$

$$D = \{v_1, v_2, \dots, v_l\}$$

Each domain $v_i = \{v_{i1}, v_{i2}, \dots, v_{ik}\}$ where $k = |v_i|$

C = set of *hard*-constraints on subsets of variables in Z , restricting the values that they can take together.

We can demonstrate these ideas with a simple example.

$Z = \{1, 2, \dots, 10\}$

$D = \{v_1, v_2, \dots, v_{10}\}$ where $v_1 = v_2 = \dots = v_{10} = \{1, 2, 3, 4, 5\}$

$C = (\forall i, j, h)(v_i = v_j \neq v_h)$

g = See Section 2.2.3.

Constraint C requires that there are only two of each value represented in the string.

An example string which represents the PCSP:

string position - PCSP variable	1	2	3	4	5	6	7	8	9	10
solution tuple - real coded	1	2	3	4	2	5	4	1	3	5

This is a general representation, suitable for any problem which is an instance of a PCSP. Each string thus formed represents a solution tuple to a PCSP when the hard-constraints are satisfied and is a single point in the search space. How we interpret the representation depends upon the form of the objective function.

2.2.3 Objective Function

The GAcSP objective function maps each PCSP solution tuple to a numeric value. The goal of the GAcSP is to find optimal or near optimal solutions to the PCSP. The GAcSP seeks to find a minimum or maximum value as defined by the objective function which satisfies the soft constraints.

For the GAcSP to be successful it is important that the optimal solution can be located in a part of the search space which is approached through sub-optimal groups of string elements (building blocks). Otherwise, the GAcSP could find the problem misleading and *GA*-hard.

GAcSP implements specialist operators of initialisation, reproduction, and crossover which are

described in detail in the following sections.

2.2.4 GAcSP Initialisation Operator

2.2.4.1 Initialisation Function

At the start of the GAcSP run, a population of strings is created by the initialisation operator. Each string is constructed by randomly selecting a value from each PCSP variable domain, and appended to generate a finite linear sequence of values. If the standard PCSP representation defined earlier is used, the order of the variables is fixed during the GA run and is denoted by string position.

Starting the search with a randomly generated population of strings provides an important test of the GAcSP ability to improve the fitness of strings over successive generations. The random population of strings ensures that the GAcSP has an opportunity to reach (i.e. explore) as much of the search space as possible. This is more important when real-coded alphabets are used to represent problems because there are more element values to express at each string position. Techniques can be employed which use heuristics to seed the initial population but this can be dangerous, causing early convergence. The initial population diversity is necessary to maintain *grist* for the *genetic mill* [Grefenstette, 1987].

Diversity is an important dynamic property of the population, and is a measure of the differences between strings in the population. Subsequent action by the reproduction and crossover operators will change the initial population diversity. As GAcSP progresses specific patterns will emerge from the random initial population of strings due to the creation of building blocks. These patterns will be dependent upon the selective pressure of reproduction and the action of the crossover operator. If the GAcSP is successful in the search, the dominant string patterns which emerge should ensure convergence to optimal or near optimal solutions.

2.2.5 GAcSP Reproduction Operator

The reproduction operator guides GAcSP through the search space by selective control using a sampling bias based upon the string fitness. The reproduction operator consists of two functions: Elitism and Reproduction.

2.2.5.1 Elitism Function

The first stage of the reproduction operator implements a technique called "elitism" [De Jong, 1975]. Elitism copies a number of the greatest fitness strings for a maximising objective function or lowest for minimising objective function, into a mating-pool population (*matepool*). Parents are randomly selected from the matepool by the crossover operator. Matepool has the same number of string positions available as the population. Elitism guarantees that the "elite" members of the population will survive into the next generation. (So long as the number of offspring required each generation is less than the (population - elite copies).) These elite strings are considered important because they will have promising string values or groups of string values (*building blocks*) to pass onto their offspring.

2.2.5.2 Reproduction Function

The second stage of reproduction fills the remaining matepool positions after the elitism function, using a biased fitness selection from the population. The biased fitness technique is *roulette wheel* selection [Goldberg, 1989], where offspring are created in proportion to their parental fitness, relative to the population average fitness.

Although the *roulette wheel* selection method is a high variance process [Goldberg, 1989], it is simple to implement, efficient, and can help speed convergence. Any sampling method based upon a finite population of finite length strings will suffer from variance, because the schema theory is based upon schema fitness averages *in the limit* which are impractical for the GA to calculate.

2.2.6 GAcSP 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 information already contained in the population. The crossover operator should allow the offspring to inherit building blocks from its parents. The crossover operator consists of three functions: UAX, Repairer and Hill-Climber.

2.2.6.1 UAX Function

The GAcSP uses a new crossover mechanism - the uniform adaptive crossover (UAX), which uses an extended string representation. The UAX mechanism allows the selection of multiple crossover points along the parent strings with those points which improve the fitness determined adaptively as the search progresses. An example of the extended string representation is shown in the following string:

extended string -	1	2	3	4	2	5	4	1	3	5
binary template	0	1	0	0	1	1	0	0	1	0

Each member of the population will have this extra binary string, so the length of each string member will be doubled to account for this extra information. This extra binary string acts as a template to control the creation of the offspring string during the crossover process. The first stage of the crossover operator is to randomly select two strings to be used as parents from the matepool. The UAX operator implemented in the GAcSP constructs a single offspring using the following steps:

Step.1 - Randomly select one of the two parent strings, call them parent 1 and parent 2. parent 1 is selected to be the starting parent to copy from in Step.2.

string position -	1	2	3	4	5	6	7	8	9	10
parent 1 string -	1	2	3	4	2	5	4	1	3	5
binary template	0	1	0	0	1	1	0	0	1	0
parent 2 string -	2	1	3	2	4	1	5	3	4	5
binary template	1	0	1	0	0	1	1	0	0	1

Step.2 - The following operations are carried out sequentially, from left to right at each binary position in turn:

Examine both parent binary values at string position 1. If both parents have the same binary values at position 1, either 0 or 1, change to copy from the other parent, (e.g. parent 1 to parent 2).

If both parent binary values at string position 1 are different then leave the copying parent unchanged.

Example:

string position -	1
parent 1 string	1
	0
parent 2 string	2
	1.

In the example above, the parent binary values are different ('0' and '1' respectively) and therefore the parent used to copy from remains unchanged. Copy the first string element '1' from parent 1 and copy the first binary element from parent 1 '0'.

Step.3 - Repeat Step.2 for all binary positions in the offspring construction sequence. For example, for string position 2 the binary bits of parent 1 and 2 are '1' and '0' respectively, so the offspring continues to copy from parent 1. The complete offspring is shown below:

from parent -	1	1	1	2	2	1	1	2	2	2
offspring 1 string -	1	2	3	2	4	5	4	3	4	5
	0	1	0	0	0	1	0	0	0	1

Step.4 - Offspring replaces the greatest fitness member of the population for a minimising fitness function (least fitness for a maximising fitness function).

Step.5 - Repair offspring to ensure it is feasible, using the greedy repair function explained below.

Step.6 - Evaluate offspring.

Step.7 - Apply optional Hill-Climber.

The technique of using the extended representation is to allow the adaptation of crossover points. The binary crossover template provides a way of recording crossover points which correspond to above average string values and allows the offspring to inherit them. One effect

of the crossover operator upon the representation, is that offspring created will not always satisfy the hard constraints. Therefore, each offspring will need to be repaired using a repairer (greedy in this case).

2.2.6.2 Greedy Repair Function

A Repairer ensures the hard-constraints are satisfied by changing individual string values. The hard-constraints used in this implementation is to satisfy the number of each PCSP value expressed in the string. The repair mechanism is to first randomly locate a string value which has more than the required constant number of representations. Next locate all PCSP values which have less than the required constant number of representations in the string. Change the over-represented PCSP value to the under-represented PCSP value which increases the string fitness. From this random starting position a sequential search is made for all over-represented values, until the representation constraints are satisfied.

One flexibility offered by GAcSP is the use of domain specific repair functions, which can be written to capture different hard-constraint relationships. For example, binary constraints can be used but may have to be restricted to those which are independent of each other (i.e. stable sets). A number of important caveats need to be considered when designing alternative repair functions. The time taken to repair a string may make the GAcSP intractable. This can happen in the case of binary constraints, when they interact and a large search space is created. Also, altering the offspring string too much may disrupt any building blocks which have been inherited from the parents.

For example, the greedy repair function can repair the offspring string to satisfy the hard constraints. The hard constraints in this implementation restrict the number of each PCSP value expressed in a string. However, this is only one possible relationship which may be used to restrict the search space. The following steps are carried out by the repair function,

Step.1 Randomly locate a string value which is over represented.

Step.2.1 Starting from the Step.1 selected string value, search sequentially (left to right) and locate a PCSP value which is under represented.

Step.2.2 Swap the value from Step.2.1 with that from Step.1 and calculate the string fitness.

Step.2.3 Return to Step.2.1 unless all under represented PCSP string values have been found or the end of the string is reached.

Step.3 Continue with Step.1 until all PCSP values are correctly represented.

For example, consider the offspring string created by the crossover operator:

string position -	1	2	3	4	5	6	7	8	9	10
offspring 1 string -	1	2	3	2	4	5	4	3	4	5
binary template	0	1	0	0	0	1	0	0	0	1

In offspring 1, the string values at positions 5, 7 and 9 are the same (e.g. $S_5 = S_7 = S_9 = 4$) yet constraint C requires that there are only two string values equal to 4. So one of these string values will need to be changed. The Repairer will randomly select one of the string positions $i = 5, 7$, or 9 . Next, find any PCSP values which are under represented. For example, because in the offspring there is only one $S_1 = 1$ instead of two, it is under represented. Then the string value at position 7 $S_7 = 4$ is changed to $S_7 = 1$ giving the following repaired string:

offspring 1 string -	1	2	3	2	4	5	1	3	4	5
	0	1	0	0	0	1	0	0	0	1

2.2.6.3 Optional Hill-Climber Function

The GAcSP strategy incorporates an optional HC in order to improve on the quality of solutions. The HC is optional because it may be used to improve on solution quality if run-time is available. It also allows some control over speed of convergence by either increasing or reducing diversity. Although the GA is a powerful heuristic technique for finding near optimal solutions to problems, it lacks a local improvement ability. By combining the GA with a HC we hope to provide this local improvement ability. We have already mentioned the advantage of using the HC with the GAcSP in terms of increasing the optimality of strings when using

real-coded alphabets (Section 1.4.3.2). The HC can also be regarded as a *super mutation* operator, in that it can increase the string fitness and put pressure on the GAcSP by increasing the average population fitness to seek improvements. Another aspect of the HC as a super mutation function is to encourage diversity which will begin to decline as the search progresses. Standard mutation operators can also increase the diversity of populations as the GA search progresses, but at the cost of destroying important building blocks. The HC should maintain and even improve upon the building blocks in a population.

The HC is a simple string element exchange function, for swapping high cost PCSP values with any other PCSP values which will reduce the string fitness. The high cost PCSP values are those which can be identified as contributing to the string fitness. A starting point is randomly selected, from which high cost PCSP values are located and swapped for any other PCSP values which minimise the string fitness. This process is repeated until no more improvement is possible, or a pre-set time limit is reached. The HC undertakes the following steps:

Step.1 Randomly locate a PCSP value in the string which has a low fitness.

Step.2.1 Starting from the value following the Step.1 selected string value, swap the value with that from Step.1.

Step.2.2 Calculate the new string fitness. Save fitness and value if greater than best fitness string recorded so-far.

Step.2.3 Continue with Step.2.1 until all string values are tested.

Step.3 Swap the best fitness value from Step.2.2.

Step.4 Continue with Step.1 until no more improvement in fitness can be achieved or until a pre-set time limit is reached.

The HC, by swapping a string value in one string position with string values in other positions, effectively tests all PCSP values at that position. This HC action will work in a similar way to the adjacency mutation or "creep" operator of Davies [Davis, 1991]. This very localised operator could assist GAcSP in overcoming a search space coding which prevents it from

reaching the global optimum.

2.2.7 GAcSP Parameters

There are a number of parameters which are set prior to the GAcSP run. GAcSP parameters include population size, number of parents, number of offspring generated, number of GAcSP cycles, GAcSP time limit, HC time limit, and number of elite string members copied. Research has established optimum parameter settings for the canonical or standard GA using a binary representation tackling specific function optimisation problems. When using a non-canonical GA with a real-coded representation there are no *a priori* reliable settings established by research. There is also a paucity of theory into the interaction between the real-coded representation and GA operators. In this situation the practical approach is to consider the following issues in deciding on preliminary initial parameter settings.

What computing resources are available in terms of processing power and memory ?

What computing resources are required by the GAcSP ?

What size and complexity of problems are to be tackled ?

What time is available to achieve satisfactory solutions ?

What quality of solutions are required ?

These questions impose limitations upon the possible parameter settings. With limited computer resources available, a balance has to be achieved whereby the parameter settings are appropriate to the problem to be tackled and ensure the results obtained are satisfactory.

Population size is a constant, pre-set before the start of the GAcSP run. The population size will depend upon the actions of the operators and problem size. A larger population will be diverse and contain many search space points for GA processing, but require more computing memory and evaluation time to process. Although smaller populations require less memory and processing, they require special operators for the search to be adequate. A population size should be large enough to represent the PCSP values in all string positions in the initial

population. Population size also has an important impact on the control of premature convergence [Booker, 1987] but must be balanced by the computational work required. We considered the results of previous GA work involving population sizing on CSOPs [Tsang and Warwick, 1989].

The number of offspring created in each complete GAcSP cycle is pre-set before the run. The number of offspring generated each cycle controls the rate of exploration carried out by the crossover operator.

A constant number (i.e. a percentage of the total population) of the best members of the population are copied directly into the matepool. This technique improves the off-line performance of the GAcSP, increasing the chance of more optimal solutions.

Parameter settings are important, but if the GA implemented is robust the impact of changes may not be considerable [Davidor, 1991]. Parameters can usually be tuned empirically and tested to improve the performance of the GAcSP.

2.2.8 Population Dynamics

The external dynamics of the population describe how the population of strings is manipulated by the GAcSP. We have already seen in Section 2.2.4.1 that the initialisation operator randomly creates a population of strings, before the main GAcSP operators are put into action. We can describe the action of the main GAcSP operators upon this initial population of strings.

2.2.8.1 External Dynamics

The main *cycle* of GAcSP operators add strings, delete strings (by replacement and non-selection) and changes individual string elements in the population. The elitism function in the reproduction operator copies the highest fitness members of the population into the matepool with the remaining positions taken by biased fitness selection from the population. The crossover function in the crossover operator creates an offspring by exchanging string elements between two parents and the offspring replaces the lowest fitness member of the population. This continues until the pre-set number of offspring is reached. This constant number of offspring controls the number of population members replaced by offspring in each generation. If this constant was

set equal to the population size, a new population would be created each generation (termed *non-overlapping* or *generational*). However, when the setting is less than the population size, some of the current matepool members will survive into the next generation. The Repairer and HC in the crossover operator alter individual string elements in much the same way as a mutation operator.

2.3 Summary

The heuristic strategy which we call GAcSP is distinguished from the standard or simple GA by the integration of reproductive best fitness string selection, an adaptive template type crossover, hill-climbing, and string repair. The design of GA operators are influenced by representation and PCSP constraints.

The reproduction operator guides GAcSP through the search space by selective control using a sampling bias based upon the string fitness. The reproduction operator first copies a number of the greatest fitness strings (i.e. *elite*) for a maximising objective function into a mating-pool population (*matepool*). Next, it fills the remaining matepool positions using a biased fitness selection from the population. Elite strings are considered important because they will have promising string values or groups of string values (*building blocks*) to pass onto their offspring.

Binary-coded representations are supported by established GA theory and operators with their perceived advantages being the opportunity to use traditional GA operators and support from the Schemata Theorem. However, drawbacks to using binary-coded representations include string length, problems such as hamming cliffs and slower convergence rates. Also, the need to decode and encode could increase the computational work load of the GA and the inverse mapping precision of the binary integers into the real values (i.e. PCSP values). A real-coded or "natural" representation results in a more compact string, does not suffer from hamming cliffs and allows us to work directly with the PCSP domain values.

Constraints are an essential feature of the PCSP and their interactions influence the design of the GA crossover and mutation operators. Constraints represent interacting relationships between

PCSP variables and can give rise to the phenomena of *epistasis* in two separate ways: *Hard* constraints must be satisfied and have a direct effect on the structural search space. *Soft* constraints are the user requirements or the constraints of the problem which can be optimised and can be incorporated into the objective function which influence the search through the fitness function.

GAs can handle hard constraints in a number of ways, by hiding them using a specially designed representation and set of operators, incorporating into the objective function using a *penalty function* or repair strings violating hard constraints. With the penalty function approach the search space becomes the space of all possible solution states, rather than solution states where the hard constraints are satisfied. Its disadvantage is in allowing the GA into infeasible parts of the search space, where solution states do not satisfy the hard constraints, and the difficulty of estimating the measure of penalty cost which should be associated with constraint violation. Strings created by GAcSP are *repaired*, ensuring that the hard constraints are satisfied. This approach has the advantage that it can prune some of the search space by preventing the GA from entering infeasible areas and eliminates difficulties associated with estimating violation costs. Also, we can make the GA strategy more robust by allowing for the possibility of incorporating different hard constraints without needing to change the general strategy. The soft constraints of the problem can be directly optimised through the objective function. Soft constraints can include resources, customer requirements, economic objectives or a combination of these and other costs. Unlike hard constraints, soft constraints need not necessarily be completely satisfied for a solution to be acceptable. The GA approach is to find optimal or near optimal solutions to PCSPs which maximise or minimise the soft-constraint violation measured through the fitness function.

Constraint interactions determine the effectiveness of the crossover operator which must have the ability to propagate building blocks throughout the population and to create new ones. Constraints create a potential difficulty where string elements which interact could do so over any distance along the string, preventing the building blocks (short defining length schemata) from developing. The *uniform crossover* developed by Syswerda [1991] is a multi-point crossover operator which may devastate tightly packed schemata but can perform better on schemata

which are not. We have adapted the uniform crossover and called it the *uniform adaptive crossover* (UAX). The UAX operator allows us to cut the string into sections along the string length as a way to retain and propagate longer schemata, preventing them from being destroyed. The points at which UAX cuts the parent strings during recombination are determined by the parent binary templates. A binary template for each string consists of a set of binary values, where each binary value is associated with a particular string value. The binary templates record successful cut-points and effectively creates an underlying binary space which gives an offspring the opportunity to inherit these building blocks utilised through the UAX crossover operator. By manipulating binary template space values, important real-coded building blocks will be inherited, which will lead the GA towards "optimal" solutions.

Although the GA is a robust technique for finding near optimal solutions in combinatorial search spaces, it generally lacks a local improvement ability. This local ability is provided by combining the GA with a simple hill-climber (HC). HC is a simple string element exchange function, for swapping high cost PCSP values with any other PCSP values which will reduce (for minimisation problems) the string fitness. After UAX, strings are repaired and hill-climbed. These repair and hill-climb functions act as a mutation operator altering individual string elements. A localised HC can increase population diversity and compensate for the possible loss of performance through the use of large alphabets. Also, the action of the HC will put pressure on the GA to seek further improvements because the population average fitness will increase as each string fitness improves. GAcSP can use domain specific repair functions, which can be written to capture different hard-constraint relationships.

Chapter 3 Tested Domains

We have tested GAcSP on the following problems which are covered in the relevant sections:

- 3.1 Processors Configuration Problem (PCP)
- 3.2 Car Sequencing Problem (CarSP)

3.1 The Processors Configuration Problem (PCP)

3.1.1 Definition

- **Definition 3.1 (Processors Configuration Problem (PCP))** The PCP is the linking together of a finite set of k independent processors, each with a fixed number of links into a multiprocessor network. The objective of the PCP is to minimise the processor paths between source and target processors, where for each k source processor there are $k - 1$ target processors.

Note: in the above definition k is equivalent to the number of PCSP domain values.

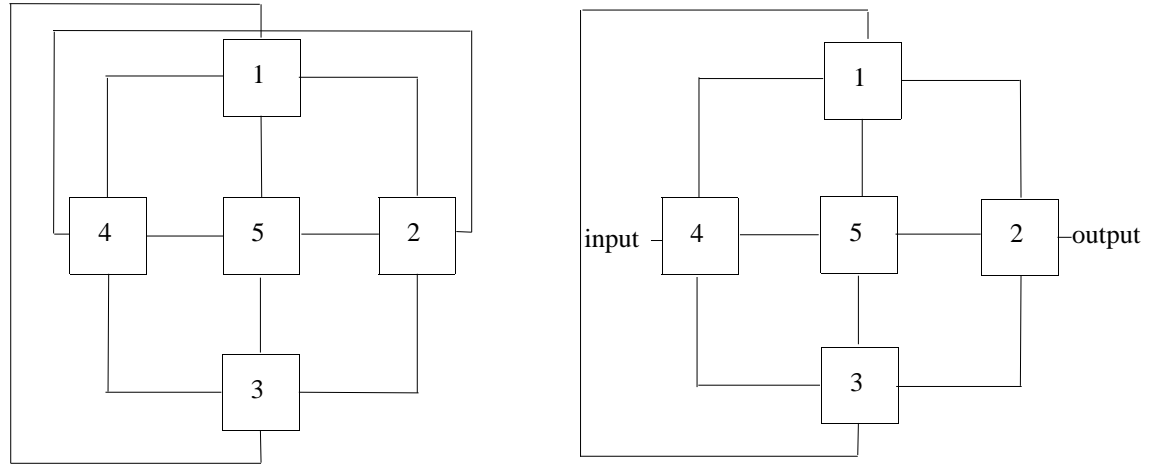
- **Definition 3.2 (Processor valency Δ)** The processor valency Δ is the fixed number of links available on each processor for connection to other processors.

Motivated by the desire to exploit transputer technology, all PCPs tackled in this thesis have processors with valency $\Delta = 4$.

A restriction on the configuration of the network is that two processor links are assumed to be used by the *system controller*, which acts as input/output to the network. Figure 3.1 (a) shows a five processor network without input/output. In this research we only consider networks with input/output, such as the network in Figure 3.1 (b).

- **Definition 3.3 (Regular Network)** A regular network is a configuration of fixed valency processors which is symmetrical about at least one axis (e.g. tori and hypercubes).

Figure 3.1: (a) Network without input/output (b) Network with input/output



- **Definition 3.4 (Irregular Network)** An irregular network is a configuration of fixed valency processors which has no guarantee of symmetry about any axis.

The irregular networks that are formed provide useful maps for connecting processors together in distributed memory MIMD machines, and for many applications where performance can exceed that of regular networks [Prior et al., 1989].

3.1.2 Graph Theory

A multiprocessor network can be described in terms of graph theory [Chalmers and Gregory, 1992] where the processors are the nodes and its links bidirectional arcs. Graph theoretic distances can provide useful measures for the irregular graphs which are formed. These measures include the diameter, and mean internode distance which can represent limiting factors on communication speed.

- **Definition 3.5 (Diameter d_{\max})** The diameter d_{\max} of a graph [Chalmers and Gregory, 1992] is an invariant, and is the maximum of the direct distances between any two nodes in the graph - where the direct distance is taken to be the minimum number of links which have to be traversed in communication between a source and target node.

- ° **Definition 3.6 (Mean internode distance d_{avg})** The mean internode distance d_{avg} of a graph [Chalmers and Gregory, 1992] is the average of the *direct distance* between any two nodes in a graph.

The mean internode distance for a graph can be calculated as follows:

$$d_{avg} = \frac{\sum_{p=1}^k \sum_{d=1}^{d_{max}} (d \cdot k_{pd})}{k^2} \quad (3.1)$$

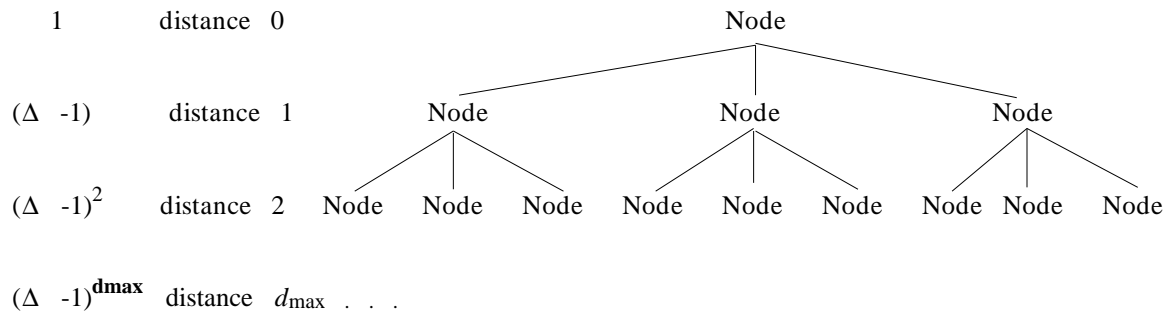
where k is the number of nodes used and k_{pd} [Chalmers and Gregory, 1992] is the number of nodes distance d away from node p .

A theoretical upper bound n_{max} on the maximum number of nodes in a graph for a fixed diameter can be calculated.

- ° **Definition 3.7 (Upper Bound n_{max})** The upper bound n_{max} for a graph is the maximum number of nodes for a fixed diameter d_{max} (assuming two arcs are used as input/output), is calculated as follows:

$$n_{max} = 1 + (\Delta - 1) + (\Delta - 1)^2 + \dots + (\Delta - 1)^{d_{max}} \quad (3.2)$$

The theoretical upper bound is based on the assumption that each node in an k node graph has the minimum distance possible to every other node. This set of routes for a node is called its *spanning tree* [Chalmers and Gregory, 1992]. We can demonstrate this assumption and spanning tree by considering a single controller connected node configuration at each diameter distance up to d_{max} as follows:



However, Equation 3.2 represents the set of minimum distance routes for a node which is connected to a system controller (i.e. $(\Delta - 1)$). In a graph where two arcs are used by the system controller only two nodes in such a graph will have this spanning tree. The other $k - 2$ nodes will have spanning trees with nodes at distances up to d_{\max} shown by equation,

$$n_{\text{Moore}} = 1 + \Delta + \Delta(\Delta - 1) + \Delta(\Delta - 1)^2 + \dots + \Delta(\Delta - 1)^{d_{\max}-1}. \quad (3.3)$$

Graphs which have k nodes with spanning trees represented by Equation 3.3 (i.e. no controller) are called *Moore* graphs. Since Moore graphs consist of nodes which have no system controller connections (i.e. Equation 3.3) they will therefore have a greater upper bound than for optimum graphs for a given diameter d_{\max} .

$$n_{\text{Moore}} > n_{\max}.$$

Since only two nodes in a graph will be connected to the system controller the n_{\max} upper bound must be regarded as not being an accurate estimate for the maximum number of nodes for a graph with a diameter. Instead, the upper bound n_{Moore} is a better approximation to the maximum number of nodes for a given diameter than n_{\max} . Furthermore, Chalmers and Gregory [1992] state it has been shown [Biggs, 1974] that except for when $d_{\max} = 1$ or $\Delta = 2$, Moore graphs can only exist for the cases: $d_{\max} = 2$ and $\Delta = 3, 7$, or 57 . The upper bound n_{Moore} with $\Delta = 4$ is therefore unobtainable. Any graphs which are formed can only approach the upper bound n_{Moore} .

The diameter d_{\max} and mean internode distance d_{avg} are measures of the compactness of a network. Three benefits of compact networks are:

- (a) The more compact the network the shorter the distances for communication, and the less consumption of network links. Compact networks with distributed loading can sustain higher levels of communication.
- (b) Compact networks involve shorter internode distances so fewer processors are interrupted from performing useful computations.
- (c) Communication between two processors involves intermediate processors propagating messages.

Compact networks can reduce the latency of communication due to propagation delays.

- ° **Definition 3.8 (Optimum Graph)** An optimum graph is where each node has the minimum possible direct distance to every other node and two nodes are connected to the system controller.

We can derive a theoretical lower bound D_{avg} for the mean internode distance of *optimum graphs*, providing an important measure for comparison with the mean internode distance d_{avg} of graphs. In order to calculate the theoretical lower bound of an optimum graph, we need to consider spanning trees of nodes with $\Delta = 3$ and $\Delta = 4$. Source nodes which are connected to the system controller will have 3 arcs for connection to other nodes, each of these 3 nodes at distance $d = 1$ will in turn have 3 arcs. Similarly nodes which are not connected to the system controller have 4 arcs for connection to other nodes, at distance $d = 1$ these each 3 nodes will in turn have 3 arcs. Therefore, the number of arcs of the source node available for connection to other nodes will determine the maximum number of nodes for a graph.

For a node p with n_{arc} arcs (where n_{arc} could be 3 or 4), the maximum number of nodes M_{pd} that it can reach in distance d is:

$$M_{pd} = n_{arc} \cdot 3^{d-1} . \quad (3.4)$$

We can calculate the number of nodes M_{pdmax} distance d_{max} away from p as

$$M_{pdmax} = (k-1) - n_{arc} \sum_{d=1}^{d_{max}-1} 3^{d-1} . \quad (3.5)$$

We can use Equation 3.4 to calculate the maximum number of nodes at distances $d = 1, 2, \dots, d_{max}$ for source nodes, where $n_{arc} = 3$ and 4. However, we need to consider the fact that k nodes of an optimum graph may be less than the total number of nodes calculated using Equation 3.4 when $d = 1, 2, \dots, d_{max}$. In which case, we use Equation 3.4 to calculate the maximum number of nodes at $d = 1, 2, \dots, d_{max}-1$ and Equation 3.5 for the additional nodes at distance d_{max} . We obtain the total distance for all paths from a single source node by multiplying the maximum number of nodes M_{pd} at each distance by the distance value and

finally adding the M_{pdmax} nodes. This calculation is carried out for each of the k nodes of an optimum graph. The lower mean internode distance bound (defined below) for an optimum graph is the total of the distance of the paths from each source node, divided by the total number of node paths k^2 . The following definition concludes these results.

° **Definition 3.9 (Lower mean internode distance bound D_{avg})** The lower mean internode distance bound D_{avg} for an optimum graph with k nodes can be calculated as

$$D_{avg} = \frac{\sum_{p=1}^k \left(\sum_{d=1}^{d_{max}-1} (d \cdot M_{pd}) + M_{pdmax} \right)}{k^2} . \quad (3.6)$$

Chalmers and Gregory [1992] provide mean internode distance d_{avg} results from their PCP program (AMP), for PCPs of 32 processors and 40 processors. The maximum number of processors they have configured for a processor valency $\Delta = 4$ and diameter $d_{max} = 3$ network is 32, whilst the theoretical upper bound $n_{max} = 40$. The GAcSP approach is to directly optimise the diameter as a constraint and measure the mean internode distance achieved.

Although Prior *et al.* [1989] use a GA to tackle the PCP their results are not directly comparable, because we have assumed that two processor links are used by the system controller.

3.1.3 Representation

From the CSP definition given in Section 1.3 and its solution tuples we can define a string representation where, each PCSP variable relates to a string position and at that string position, has an assigned value taken from the variable domain. This representation is limited to even valency (Δ) processors.

° **Definition 3.10 (PCSP(PCP))** We formally define the PCSP(PCP) as:

$Z = \{1, 2, \dots, N\}$ where $N = k \cdot z$ and $z = \Delta/2$

$D = \{v_1, v_2, \dots, v_N\}$ where $v_1 = v_2 \dots = v_N = \{1, 2, \dots, k\}$

$$C = (\forall i, j, h)(v_i = v_j \dots = v_z \neq v_h)$$

Constraint C requires that there are only two of each processors in the string.

g = The objective function g is defined in Section 3.1.4.

For example, a 5 processor PCP solution tuple could be represented as:

string position - PCSP variable	1	2	3	4	5	6	7	8	9	10
solution string - PCSP value	1	2	3	4	2	5	4	1	3	5

In this example, each PCSP variable domain (i.e. {1, 2, 3, 4, 5}) represents the set of processors to be configured by the GAcSP. We can see from the example string that there are two of each processor, numbered 1 - 5 represented. Each string element represents 2 processor links to element values either side. For example, processor 2 is linked to processors 1, 3, 4, and 5 taking up the four links available. For all strings, the first and last string element processors have three links available for connection to other processors (except when the first and last element represent the same processor), because each uses one link to connect to the system controller. The first string element processor 1 and last string element processor 5 in the above example, are linked to the system controller. Because representation elements depend upon their neighbouring values, the objective function of the PCP representation does not depend on the absolute position of values. This representation is compact and ensures that all processor links will be used, because each string element will automatically have neighbours. From this description we can see that the representation is limited to PCPs with even valencies. (If the string was regarded as forming a ring where the string ends are joined together, then it could represent PCPs without a system controller.)

3.1.4 Objective Function

The GAcSP objective function maps each PCP solution tuple to a numeric value which we call the *fitness*. The goal of the GAcSP is to find an optimal or near optimal solution tuples to the PCP. The GAcSP seeks to find a *minimum* fitness as defined by the objective function. We set a value for a numerical integer constant $c \leq d_{\max}$ as the maximum diameter constraint for a graph. The following objective function measures the total distance for all nodes at greater

distances than the diameter constraint c .

$$\text{minimise } fitness = \sum_{p=1}^k \sum_{d=(c+1)}^{d_{\max}} k_{pd} \cdot (d - c) . \quad (3.7)$$

where k is the number of nodes used and k_{pd} is the number of nodes distance d away from node p .

The objective function fitness is based on the processor to processor distances because the difference in fitness values (i.e. standard deviation) between strings will be greater than for the mean internode distance. GAcSP is given detailed knowledge of processor paths which require local improvement and the GA component has more accurate fitness values for biased selection for mating. Also to achieve optimum graph configurations each node must have minimum paths to every other node, and therefore search pressure needs to be placed on improving node to node paths. We can see from Equation 3.6 that the fitness is effectively a total measure of all the processor to processor distances, greater than the diameter constraint c . The value of c can provide GAcSP with useful knowledge in guiding GAcSP by acting as a filter to focus the search towards areas of possible improvement. Many paths in a PCP configuration will be equal to 1 - these are the nodes directly connected to other nodes. Obviously, nodes at distance 1 cannot be improved any further with nodes at distance 2 providing more room for improvement. If we set $c = 2$ we filter out these direct connections and concentrate the search on improving all other paths. When a processor in a network configuration has self links or is connected more than once to the same processor, the effect is to increase the string fitness and penalise these strings. Self connections, and double connections are links which could otherwise reduce a number of the processor to processor paths. As well as the fitness, the mean internode distance is calculated using Equation 3.1 and recorded on the string.

3.2 The Car Sequencing Problem (CarSP)

3.2.1 Definition

- ° **Definition 3.11 (Car Sequencing Problem CarSP)** The CarSP has a set of predefined car types, each requiring a different set of options (e.g. car radio, seat covers etc.) fitted by

specialist workstation teams on an assembly line. The task is to sequence a specified number of cars for each car type ensuring the workstation capacity is not exceeded or capacity violations are minimised.

- **Definition 3.12 (Schedule)** The schedule is a linear sequence of cars moving at a constant speed through the assembly line of workstations.

In a car production unit, cars are required to pass through an assembly line of specialist workstations in order to have options fitted on them, such options could include car radios, sunroof and furry dice. Each workstation team is required to fit options to cars whilst the cars travels through its workstation and finishing before each car leaves. Workstations have been designed so that there is sufficient time for teams to fit a maximum number of options (*capacity*) to consecutive cars requiring them. Groups of cars which share the same option requirements are sequenced to produce a single schedule which does not exceed the option fitting capacity of each workstation. We first formally define the car sequencing problem (CarSP) and provide an example of a CarSP where a schedule can be generated which satisfies the workstation capacities. In the next section we consider an approach to tackle a CarSP which cannot produce a schedule satisfying workstation capacities.

In a CarSP there are k car types (equivalent to the number of PCSP values) which share the same option requirements. Each of the k car types has pr_j cars and these are the *production requirements* of the CarSP. For an n option CarSP with k car types we define an $n \times k$ *option requirement* matrix, where $o_{mj} = 1$ if option m is required by car type j , otherwise $o_{mj} = 0$.

Grouping the different option requirements into car types can reduce the size of the CarSP. Given N cars to be scheduled, grouped into k car types, the complexity of the CarSP reduces from N^N to N^k , ($k \leq 2^n$).

The total number N of cars to be sequenced can be calculated using Equation 3.8:

$$N = \sum_{j=1}^k pr_j. \quad (3.8)$$

For each option m there is a specialist workstation on the assembly line with a team of workers which can fit p_m options in the time it takes for q_m cars to pass through its workstation. The ratio p_m/q_m represents the workstation m capacity constraint. We can calculate the total number O_{num_m} of option m required in the schedule, using Equation 3.9:

$$O_{num_m} = \sum_{j=1}^k p_{mj} \cdot o_{mj}. \quad (3.9)$$

Since the ratio p_m/q_m determines the maximum number of cars requiring the option in a sequence of cars, then the total number in a schedule can be calculated as

$$O_{max_m} = \frac{p_m}{q_m} \cdot N. \quad (3.10)$$

Furthermore, using the results of Equations 3.9 and 3.10 we can calculate the *utility ratio* u_m for option m , which is the degree of capacity constraint satisfaction

$$u_m = \frac{O_{num_m}}{O_{max_m}}. \quad (3.11)$$

An important conclusion from these results is that, 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 $(\forall m)(u_m \leq 1)$. Therefore, if we calculate the utility ratio for each option of a CarSP we can at least determine *a priori* whether it is definitely unsolvable or possibly solvable.

The *average utility* \hat{u} for a CarSP can provide one useful measure to compare CarSPs which have the same number of cars to sequence, the same options and capacity constraints. Also, the *average utility* \hat{u} is the CarSP average degree of capacity constraint satisfaction, and calculated as

$$\hat{u} = \frac{\sum_{m=1}^n u_m}{n}. \quad (3.12)$$

3.2.2 Example Of A Solvable CarSP

We can demonstrate these ideas with a simple example CarSP with 12 cars to sequence, 3 options each with a typical capacity constraint.

		car type					capacity constraints				
		1	2	3	4	p/q	p/q	Omax	Onum	u	
option	car radio	1	1	0	0	1	1/2	.50	6	5	.83
	furry dice	2	0	1	0	1	2/3	.66	8	6	.75
	sunroof	3	0	0	1	0	1/3	.33	4	4	1.00
cars in type		2	3	4	3						

Table 3.1: Solvable example CarSP

Table 3.1 shows the capacity constraint ratio p_m/q_m for each option, and the percentage capacity constraint ratio. Using Equation 3.10 we can calculate the number of options allowed in the schedule with no violation of the capacity constraints. These numbers are given in Table 3.1 under column 'Omax'. Equation 3.9 results under column 'Onum', give the number of cars with each option in the schedule to be sequenced. The utility ratio for each option are listed under ' u '. The average utility can be calculated for the example CarSP using Equation 3.12, giving $\hat{u} = .86$.

The following example schedule satisfies the capacity constraints in Table 3.1:

schedule position $i = 12 \ 11 \ 10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1$
 schedule $4 \ 3 \ 4 \ 2 \ 1 \ 3 \ 2 \ 1 \ 3 \ 2 \ 4 \ 3 \rightarrow$ assembly line

where the position of cars in the schedule are numbered in sequence from right to left, with car 1 as the first to enter the assembly line of workstations and car 12 the last. In general, a car i when $i = 1$ is the first car in the schedule and represents the *beginning* and when $i = N$ the *end*. It is important to recognise that the direction the schedule passes through the assembly line can make a difference to whether the capacity constraints are satisfied. As a car requiring an option enters the workstation which fits that option, the number of cars requiring m in the consecutive q_m cars following will determine if the capacity is exceeded. This is particularly relevant at the beginning and end of a schedule where a change of direction

may provide a consecutive group of cars which exceeds a workstation capacity. In the next section we shall consider in more detail, how the relative positioning of cars in a schedule can assist the workstation teams.

3.2.3 The Penalty Function

3.2.3.1 Basic Method

There are CarSPs which are not solvable, where the capacity constraints cannot be satisfied (i.e. $\text{Onum}_m > \text{Omax}_m$). In these problems, *penalty functions* [Parrello et al., 1986] are used to minimise the capacity constraint violation. The use of penalty functions can also improve the car spacing arrangements in a schedule, and this can assist the assembly line workstation teams. Penalties reflect the ability of workstation teams to cope with options exceeding the capacity constraints. Some workstation teams may be able to cope with no more than an extra option above their capacity, so extra options in the schedule are spaced apart by the use of high penalties. If we add an extra option 3 to type 1 cars in the simple example used earlier, we can make the previous example CarSP unsolvable.

		car type				capacity constraints				
		1	2	3	4	p/q	p/q	Omax	Onum	u
car radio	1	1	0	0	1	1/2	.50	6	5	.83
option furry dice	2	0	1	0	1	2/3	.66	8	6	.75
sunroof 3		1	0	1	0	1/3	.33	4	6	1.50
cars in type		2	3	4	3					

Table 3.2: Unsolvable example CarSP

In this second example $\text{Onum}_3 = 6$ (Equation 3.9) $>$ $\text{Omax}_3 = 4$ (Equation 3.10). The number of cars requiring option 3 is 6, yet the capacity constraint $p_3/q_3 = 1/3$ will only allow a maximum 4 of the 12 cars to be scheduled to have this option. The utility ratio $u_3 = (6/4) > 1$, and therefore the example CarSP is unsolvable.

For example, if we were to position the type 3 cars requiring option 3, we would not be able to position any type 1 cars requiring option 3 without violating the capacity constraint. Therefore, we need to add the type 1 cars in such a way as to minimise the capacity constraint

violation according to a penalty function. With the capacity constraint 1/3 for option 3, the two cars which follow the first requiring the option is a sub-sequence of cars defined by Parrello [Parrello et al., 1986] as the *interval of relevance*. In general, the interval of relevance for option m is equal to (q_m-1) . The penalty function assigns a penalty value depending upon the number of cars requiring the option in the interval of relevance in excess of workstation capacity q_m .

For example the following gives penalty values for extra cars requiring option 3 in the interval of relevance:

	number of cars	
	1	2
option 3 sunroof	2	5

There is no penalty if no cars in the interval of relevance require option 3, but when one car in the interval of relevance requires option 3 the penalty value is 2. Each option will have penalty values which reflect the capacity of the workstation teams to cope with work exceeding the workstation capacity p_m . In general, we can define an $n \times (q_m-1)$ *penalty value* matrix, where P_{mr} is a numeric integer penalty for each r cars requiring option m . If the number of option m required in a q_m consecutive sequence of cars is less than the capacity p_m , then the penalties will be 0 (e.g. if $r < p_m$ then $P_{mr} = 0$). Each car i in a schedule S represents the first of a q_m sub-sequence for each of its options $o_m S_i = 1$. We can calculate the *penalty cost* of option m for a car i in schedule S_i by using the value from summing the number of cars requiring option m in q_m cars following S_i in the schedule, to index the penalty matrix as follows,

$$cost_{S_i m} = P_m(o_m S_i) \cdot \sum_{l=(i+1)}^{(i+(q_m-1)) \leq N} o_m S_l \quad (3.13)$$

where $o_m S_i = 1$, if car i requires option m . It follows, that if $P_{mi} = 1$ for $m = 1, 2, \dots, n$ and $o = p_m+1, \dots, (q_m-1)$, then Equation 3.13 calculates the total penalty cost for a single car i .

For each option, it may be possible to improve the car spacing arrangement in a schedule, to assist the workstation teams installing the options. Consider the current example:

schedule position		4	3	2	1
car type	...	4	2	1	3 → assembly line
requires option 3 ?		N	N	Y	Y

with the capacity constraint of 1/3 for option 3, car 1 penalty cost for option 3 = 2. In order to encourage improved spacing of cars requiring options in the schedule, a smaller *proximity interval* I_m is defined for CarSPs along with a *proximity factor* f_m . When the capacity constraints in a schedule cannot be satisfied the proximity interval and factor are used to balance the extra work required. So improved spacing of cars requiring options can give workstation teams more time to work on them. In the above example, if car type 1 in position 2 swaps places with car type 2 in position 3 then the space between option 3 cars gives the workstation team more time to fit option 3 on the car in position 1.

If a car i option m penalty cost is greater than 0 (i.e. $\text{costS}_{im} > 0$) and a car in the proximity interval I_m requires option m then a fixed proximity factor f_m is incurred in addition to the penalty cost as follows

$$\text{TcostS}_{im} = \begin{cases} \text{costS}_{im} + f_m, & \text{if } \text{costS}_{im} > 0 \text{ and } \left(\sum_{l=(i+1)}^{(i+I_m) \leq N} o_{ml} \right) \geq 1 \\ \text{costS}_{im}, & \text{otherwise.} \end{cases} \quad (3.14)$$

In the following example option 3 proximity interval and factor, if a car within the proximity interval of 1 car requires option 3 then the proximity factor 7 is added to the car penalty cost.

	proximity	
	interval	factor
option 3	1	7

By adding the proximity factor to the current example we can see how pressure is placed on cars requiring option 3 to have other non-option 3 cars to come between them. The following example has a car at position 2 requiring option 3 within a proximity interval 1 from car 1,

and has the proximity factor of 7 added to its cost.

schedule position		4	3	2	1
car type	...	4	2	1	3 → assembly line
requires option 3 ?		N	N	Y	Y

Car 1 penalty cost + proximity factor for option 3 = (2 + 7).

By spacing apart the cars requiring the same option, the additional proximity factor is not active. With a proximity interval of 1, swap the car from position 2 (type 1) with car position 3 (type 2) to obtain

schedule position		4	3	2	1
car type	...	4	2	1	3 → assembly line
requires option 3 ?		N	Y	N	Y

Car 1 penalty cost + no proximity factor for option 3 = 2.

Adding all penalty and proximity costs for a single car i in schedule S , gives us the total car penalty cost:

$$\text{cost}S_i = \sum_{m=1}^n \text{Tcost}S_{im}. \quad (3.15)$$

These total car costs for all cars $i = 1$ to N in schedule S can be added together to give the *schedule cost*:

$$\text{schedule cost} = \sum_{i=1}^N \text{cost}S_i. \quad (3.16)$$

The schedule cost represents the sum of penalties for all cars which violate the capacity constraints and proximity intervals. A schedule can be derived from Table 3.1 where capacity constraints can be satisfied, and the *schedule cost* = 0.

3.2.4 Theoretical Lower Bound For Unsolvable CarSPs

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 conditions under which this lower bound formula is limited are for solvable CarSPs (i.e. schedule cost = 0) made unsolvable by a single option over-utilised, $\forall(P_{mr}) = 1$ for $r \geq p_m$ and $f_m = 0$.

For option m we can calculate the number of cars O_{exe_m} which exceed the maximum number of options O_{max_m} allowed in a schedule as

$$O_{exe_m} = O_{num_m} - O_{max_m}. \quad (3.17)$$

Consider the case where O_{max_m} options have been sequenced to satisfy the capacity constraint p_m/q_m , the remaining O_{exe_m} options need to be placed in the spaces so as to minimise the schedule cost. For example, if we add 2 extra options to an interval of relevance as in,

$$\begin{array}{cccccccccccccccc} S \text{ position } i & = & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ S \text{ option } m & = & 3 & & & 3 & & 3 & 3 & 3 & 3 & & 3 & \\ \text{violations } S_{cost} & = & 3 & & & & & & V & V & V & & & \end{array} \rightarrow \text{assembly line}$$

gives the minimum violation cost. This *grouping* of extra options in available spaces in each interval of relevance, ensures the minimum number of options violated. If there are $(q_m - p_m)$ option m in an interval of relevance, then the minimum number of violations for q_m options will be equal to q_m . We can calculate the number of q_m options by finding the number of spaces available in S and multiplying by q_m to give us the total minimal violation as,

$$((O_{exe_m}/(q_m - p_m)) \cdot q_m). \quad (3.18)$$

In addition, an extra option m placed in a space at the end of S presents a special case where only p_m options are violated, for example:

$$\begin{array}{cccccccccccccccc} S \text{ position } i & = & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ S \text{ option } m & = & 3 & & 3 & & 3 & & 3 & & 3 & & 3 & \\ \text{violations } S_{cost} & = & 1 & & & & & & V & & & & & \end{array} \rightarrow \text{assembly line}$$

Allowing for this special case, the lower bound formula becomes

$$\text{lower bound} = ((O_{exe_m}/(q_m - p_m)) \cdot q_m) - p_m. \quad (3.19)$$

3.2.5 Representation

GA operators manipulate artificial chromosomes in the form of string-like data structures. These data structures are constructed from elements of the CarSP. We have already seen an example data structure as a complete schedule in Section 3.2.2. We can construct a GA string composed of a linear sequence of all the cars to be scheduled. More formally we can define the schedule S position S_i , as a domain variable from the set of cars $\{1, 2, \dots, N\}$ and the domain of each variable as the set of car types $\{1, 2, \dots, k\}$.

° **Definition 3.13 (PCSP(CarSP))** We formally define the PCSP(CarSP) as:

$$Z = \{1, 2, \dots, N\}$$

$$D = \{v_1, v_2, \dots, v_N\} \text{ where } v_1 = v_2 = \dots = v_N = \{1, 2, \dots, k\}$$

$$C = (\forall i, j) (pr_j = |\{v_1, v_2, \dots, v_N = j\}|)$$

Constraint C requires that CarSP production requirements pr_j are represented in the string.

g = The objective function g is defined in Section 3.2.6.

An example string representation:

string position - PCSP variable	1	2	3	4	5	6	7	8	9	10
solution tuple - PCSP value	1	2	3	4	2	5	4	1	3	5

3.2.6 Objective Function

The GAcSP objective function maps each CarSP solution tuple to an numeric value. The goal of the GAcSP is to find tuples to the CarSP which minimise the capacity constraint violation. The CarSP solution tuple numeric value is usually called the fitness, which is calculated by adding the penalty costs for each car i in the schedule S . The following equation calculates the schedule *fitness* as

$$fitness = \sum_{i=1}^N cost_{S_i} . \quad (3.20)$$

3.3 Summary

We have tested GAcSP on the processors configuration problem (PCP) and car sequencing problem (CarSP).

The PCP is the linking together of a finite set of independent processors, each with a fixed number of links into a multiprocessor network, with the objective of minimising the processor paths between source and target processors. Although all PCPs tackled in this thesis have processors with four links we can generalise our approach to processors with an even number of links. Furthermore, we assume that two processor links are to be used by the *system controller*, which acts as input/output to the network and that irregular graphs will be formed. Graph theoretic distances can provide useful measures for these irregular graphs, which include the diameter and mean internode distance. The diameter is the maximum direct distance in terms of links to be traversed between any two nodes in the graph and the mean internode distance of a graph is the average of the direct distance between any two nodes in a graph. Both of which can represent limiting factors on communication speed. The diameter and mean internode distance are measures of the compactness of a network. Three benefits of compact networks are the less consumption of network links; fewer processors interrupted; and reduction of propagation delay.

We also derived a theoretical lower bound for the mean internode distance of optimum graphs, providing an important measure for comparison with irregular graphs. In order to calculate the theoretical lower bound of an optimum graph, we considered ideal node configurations (spanning trees) for nodes connected to the system controller and those are not connected. The GAcSP approach is to directly optimise the diameter as a constraint and measure the mean internode distance achieved. The objective function fitness for test PCPs is based on the processor to processor distances because the difference in fitness values between strings will be greater than for the mean internode distance. Also, to achieve optimum graph configurations each node must have minimum paths to every other node, and therefore search pressure needs to be placed on improving node to node paths.

The task of CarSP is to sequence a specified number of cars, each requiring a different set

of options (e.g. car radio, seat covers etc.) fitted by specialist workstation teams on an assembly line, so that specialist workstation capacity is not exceeded or capacity violations are minimised. Cars are required to pass through an assembly line of specialist workstations where each workstation team is required to fit options to cars, finishing before each car leaves. Workstations have been designed so that there is sufficient capacity for teams to fit a maximum number of options to consecutive cars requiring them. For each option workstation we defined a ratio representing the workstation capacity constraint. Furthermore, we can calculate the utility ratio (degree of capacity constraint satisfaction) using the option ratio and the number of cars in the CarSP requiring that option. The utility ratio provides a sufficient condition to prove that a CarSP is unsolvable when it is greater than 1. By calculating an average utility for a CarSP we can compare CarSPs which have the same number of cars to sequence, the same options and capacity constraints.

In unsolvable CarSPs penalty functions are used to minimise the capacity constraint violations and improve the car spacing arrangements in a schedule. Penalties are used which reflect the ability of workstation teams to cope with options exceeding the capacity constraints. For example, some workstation teams may be able to cope with no more than an extra option above their capacity, so extra options in the schedule are spaced apart by the use of high penalties. The penalty function assigns a penalty value depending upon the number of cars requiring the option in a specified consecutive number of cars (interval of relevance) in excess of workstation capacity. In addition, for each option, it may be possible to improve the car spacing arrangement in a schedule. To further encourage improved spacing of cars requiring options in the schedule, a smaller proximity interval is defined for CarSPs along with a proximity factor. When the capacity constraints in a schedule cannot be satisfied the proximity interval and factor are used to balance the extra work required. So improved spacing of cars requiring options can give workstation teams more time to work on them. By spacing apart the cars requiring the same option, the additional proximity factor does not increase the car penalty cost. The schedule cost is therefore the total car costs for all cars in schedule.

For certain unsolvable CarSPs we derived an theoretical lower bound to test the quality of GAcSP results. The conditions under which this lower bound formula is limited are for solvable CarSPs (i.e. schedule cost = 0) made unsolvable by a single option over-utilised, and proximity factors are equal to 0 and all violations are equal to 1. The theoretical lower bound calculation is based on the assumption that the maximum number of options defined by the ratio can be scheduled and that cars exceeding this number have a sequence pattern which minimises the penalty costs.

Chapter 4 Implementation And Testing Methodology

4.1 Implementation Details

4.1.1 Design Decisions

We implemented the GAcSP program GA1 in the C programming language because it provides the following qualities:

- (a) C programs can out-perform interpreted languages such as Prolog.
- (b) Performance and memory handling can be improved by the use of address pointers.
- (c) The structural quality of C facilitates modularity of code and data.
- (d) A large number of library routines are available.
- (e) It is portable to different systems.

Program performance was regarded as important because the evaluation function used by the GA and HC component of GAcSP are compute intensive procedures. The structured facility of C allows the compartmentalisation of code and data, and gives the programmer the capability to create separate independent subroutines (functions). The GA by its nature is modular, in that several, separate operators comprise its working cycle. For example in the simple GA there are operators for reproduction, crossover, and mutation. In GAcSP the repair and HC function acts as a mutation operator. The ability C has for modularity is important, particularly with regard to facilitating the development and subsequent testing of GA1.

4.1.2 Compiling And Running Programs

All programs were compiled and tested on a SUN 4/110 under UNIX 4.0 operating system.

4.2 Program GA1

4.2.1 Description

Program GA1 based upon our GAcSP model presented in Section 2.2, implemented three basic GA operators, namely initialisation, reproduction, and crossover. The relationship between these operators is outlined in Figure 4.1, and are briefly explained in the following:

- (a) The initialisation operator creates a population of strings by randomly selecting a value for each PCSP variable in sequence and appending it until every variable has a value. A fixed length string is formed, with the length of the string dependent upon the number of variables in the PCSP. The completed string is repaired using a repair function.
- (b) The reproduction operator first copies a pre-set number of the minimal fitness strings into the matepool. Then it fills the remaining matepool positions by implementing a roulette wheel selection process [Goldberg, 1989]. The roulette wheel selection process is a biased selection of minimal fitness members of the population. A selected population member is the term at which the result of subtraction between a fitness series and the population fitness is less than a product of the population fitness and random number variable.
- (c) The crossover operator randomly selects two parents from the matepool and exchanges the parents genetic material to create a single offspring. This offspring replaces the worst fitness string in the population and is then repaired and evaluated. If the HC is switched on the offspring is hill-climbed until there is no further improvement, or a pre-set time limit is reached.

4.2.2 Domain Specific Functions

For each problem tackled by GAcSP there are domain specific evaluation functions required. These functions include a function for calculating the cost or fitness of a problem solution and a function for identifying high cost string elements. (A single evaluation function can be used in some cases.) Information regarding the high cost string elements is used by the HC and repair function.

4.2.2.1 PCP Evaluation Functions

The PCP evaluation function generates a minimum node-node distance matrix by counting the number of links traversed in direct communication between each node and all other nodes. From this table of node-node distances, a mean internode distance is calculated. A separate evaluation function identifies nodes which have distances to other nodes greater than a given constant c .

4.2.2.2 CarSP Evaluation Functions

The CarSP evaluation function uses a penalty function to calculate the cost or fitness of a complete schedule. A separate evaluation function was used to identify individual cars in a schedule with capacity constraint violations.

4.2.3 Program Parameters

Population size is a pre-set constant, (see n in Figure 4.1).

The number of offspring generated each GA cycle is a pre-set constant, (see $ospring$ in Figure 4.1).

A maximum number of GA cycles for each run is a pre-set constant.

A maximum run-time in CPU seconds is a pre-set constant.

4.2.4 Population Dynamic

The initial population is randomly generated by the initialisation operator. The reproduction and crossover operators change this population during each cycle by a process of selection and replacement. Reproduction first copies minimum fitness population members into a matepool and then fills up the remaining matepool positions by a fitness biased selection. Crossover creates offspring which replace the worst fitness members of the matepool. After crossover matepool becomes the *population* for the next GA cycle.

4.3 Experimental Methodology

Much of the dissertation work was empirical - formulating and testing hypotheses. The testing strategy was to write separate operator functions and test them to measure their effectiveness. This allows the development of the GA to progress, and be directed by the quality of the test results. This approach is important because tackling the PCSP using a GA is new, making comparisons with other PCSP methods difficult. As each new operator is written and tested, each new idea or hypothesis can be systematically tested against the results of other operators.

4.3.1 Recording Results

The results obtained for each test are the total number of iterations for a run, the run-time in CPU seconds, solution or minimised solution tuple obtained (i.e. best string) and time taken to achieve the best string. Other information may be recorded on the string and returned with the best fitness string. Due to the stochastic nature of the GA, results for each experiment are recorded for a series of runs; the number of which is decided by problem complexity.

4.3.2 Terminating Conditions

4.3.2.1 GA Conditions

Conditions can be pre-set which allow the controlled termination of GA1. The following briefly describes terminating conditions:

- (a) The maximum CPU second run-time can be set, to test the GA for a fixed period of time. This terminating condition is only tested when set greater than zero.
- (b) A maximum number of GAcSP cycles may be used.
- (c) A best value solution tuple within a required percentage of a pre-set optimal (if known, approximated, or guessed) can be set. This will determine if the best minimised solution after each GA cycle is within a required percentage of the optimal value.

If none of the pre-set terminating conditions listed above have been reached, the default terminating condition of program GA1 is the test for *full* convergence of the population. Full convergence is when the fitness of all strings in the population is the same.

4.3.2.2 HC Conditions

The HC algorithm within GAcSP can have a maximum CPU second time limit specified. The time limit does not apply when no further improvement is achieved within the time limit.

4.3.3 Methods Of Analysis

Results obtained from experimental tests are recorded and tabulated into summary tables and graphs, presented with discussions in Chapter 5 for the processors configuration problem, and Chapter 6 for the car sequencing problem. Graphs, tables of data and statistical analysis techniques (two way *analysis of variance*) are used to explain the mechanisms underlying the process of GAcSP.

4.4 Summary

GAcSP program GA1 was implemented in the C programming language to due to advantages in program performance and structured facility. The performance of GA1 depends upon the compute intensive evaluation function. The structured facility of C allows the compartmentalisation of code and data, and gives the programmer the capability to create separate independent subroutines (functions). The genetic algorithm (GA) by its nature is modular, for example in the simple GA there are operators for reproduction, crossover, and mutation. All programs were compiled and tested on a SUN 4/110 under UNIX 4.0 operating system.

Program GA1 implemented GA initialisation, reproduction, and crossover operators. The initialisation operator creates a population of strings by randomly selecting a value for each partial constraint satisfaction problem (PCSP) variable in sequence and the completed string is repaired using a repair function. The reproduction operator copies a pre-set number of the lowest fitness strings (for a minimising objective function) into the matepool and then it fills the remaining matepool positions by using a roulette wheel selection process. The crossover

operator randomly selects two parents from the matepool and exchanges the parents genetic material to create a single offspring which replaces the greatest fitness (for a minimising objective function) string in the population, which is then repaired and evaluated. If the hill-climber (HC) is switched on the offspring is hill-climbed until there is no further improvement, or a pre-set time limit is reached.

For each problem tackled by GAcSP there are domain specific evaluation functions required. These functions include a function for calculating the cost or fitness of a problem solution and a function for identifying high cost string elements. Information regarding the high cost string elements is used by the HC and repair function. The processors configuration problem (PCP) evaluation function generates a minimum node-node distance matrix by counting the number of links traversed in direct communication between each node and all other nodes. From this table of node-node distances, a mean internode distance is calculated. A separate evaluation function identifies nodes which have distances to other nodes greater than a given constraint constant. The car sequencing problem (CarSP) evaluation function uses a penalty function to calculate the cost or fitness of a complete schedule. A separate evaluation function was used to identify individual cars in a schedule with capacity constraint violations.

For each set of test results we record the total number of iterations for a run, the run-time in CPU seconds, solution or minimised solution tuple obtained (for minimising objective function) and time taken to achieve these solutions. Also, program parameters can be pre-set to allow the controlled termination of GA1 these include the maximum CPU run-time, and maximum number of GA1 cycles. The default terminating condition of program GA1 is the test for full convergence of the population when the fitness of all strings in the population is the same. The HC algorithm within GAcSP can have a maximum CPU second time limit specified but does not apply when no further improvement is achieved within the time limit.

Chapter 5 PCP Results

5.1 Plan

The purpose of this chapter is to report on GAcSP tackling the processors configuration problem (PCP) as our first example PCSP. We compare the performance of GAcSP (without the hill-climbing (HC) component) with a specially written program for PCP and then we add the HC component to the GAcSP. PCPs with the following characteristics are considered under sections:

- 5.2 GAcSP Tackling Different Sized PCPs Without HC
- 5.3 GAcSP Tackling Different Sized PCPs With HC

In each section we describe the different characteristic PCPs tested. The PCPs provide a measure of GAcSP performance, and allow a comparison with available published research results. For all tests undertaken, GAcSP constants are described with their values. These include population size, number of lowest fitness members copied directly into the mating pool each cycle, number of offspring created each cycle, maximum number of cycles and run-time. In addition, the maximum improvement time allowed for each offspring for the HC components set. We summarise and present the results from the tests in table form with explanations for the terms used. Claims are made and *followed* by supporting analysis of the results, assisted by graphs where appropriate.

5.2 GAcSP Tackling Different Sized PCPs Without HC

In order to compare our results with other researchers, *Experiment 5.2* consisted of nine separate tests which were undertaken on PCPs with 32 - 40 processors. For each test the diameter constraint was set at $c = 2$. PCPs with 32 - 40 processors have a maximum diameter constraint of $d_{\max} = 3$ [Chalmers and Gregory, 1992]. Setting $c < d_{\max}$ ensures that string fitness is always positive, and makes HC work harder to improve the fitness (when switched on). Due to the stochastic nature of the GAcSP there were five runs for each test. All tests were run on a SUN 4/110 under UNIX 4.0 operating system. The GAcSP constant values of all the

tests were:

- (a) For each PCP test, the same randomly pre-generated population of 80 strings were used on all 5 runs. (Population sizes of 80 and 100 were found to provide good results in earlier research on CSOPs [Tsang and Warwick, 1989].)
- (b) 10% of the minimal fitness (elite) population strings were copied directly into the mating pool at the reproduction phase of GAcSP.
- (c) The number of offspring created each GAcSP cycle was arbitrarily set at four.
- (d) GAcSP termination conditions for each test run were set at maximum number of generations = 300, maximum run-time = 10 CPU hours.

5.2.1 Results Of Experiment 5.2

All GAcSP Experiment 5.2 results have been summarised in Table 5.1.

Table 5.1: Summary GAcSP without HC Experiment 5.2 results										
number processors k =	32	33	34	35	36	37	38	39	40	
theoretical LB D_{avg}	2.29	2.31	2.33	2.35	2.37	2.39	2.40	2.42	2.43	
AMP d_{avg}	2.31									2.53
GAcSP Results										
best d_{avg}	2.33	2.37	2.40	2.42	2.42	2.45	2.47	2.51	2.51	
max d_{eval}	4	4	4	4	4	4	4	4	4	
<i>paths</i>	12	15	24	33	31	47	49	65	62	
avg d_{avg}	2.344	2.38	2.40	2.422	2.446	2.47	2.488	2.516	2.524	
avg time best <i>sec</i>	797	1101	1018	1557	1579	1552	2620	2964	4549	
avg run-time <i>sec</i>	1290	1450	1701	2401	2328	2967	3810	4408	5143	

Table 5.1 explanation.

number processors k	= - k processor PCP.
theoretical LB D_{avg}	- The theoretical lower mean internode distance bound for optimum graphs (see Equation 3.6).
AMP d_{avg}	- The best mean internode distances for the AMP program [Chalmers and Gregory, 1992].
best d_{avg}	- The best mean internode distance (see Equation 3.1) of the 5 runs.
max d_{eval}	- The maximum direct processor to processor distance achieved.
$paths$	- The total number of processor paths greater than maximum diameter constraint $d_{\text{max}} = 3$ for 32 - 40 processor PCPs.
avg d_{avg}	- The average mean internode distance of the 5 runs.
avg time best sec	- The average time in CPU seconds taken by the GAcSP to achieve the best mean internode distance.
avg run-time sec	- The average of all run-times (GAcSP termination) in CPU seconds taken for each processor test.

5.2.2 Experiment 5.2 Discussion

° **Claim 5.1** The GA component of GAcSP can provide consistent near optimal diameter constraint satisfaction results to 32 - 40 processor PCPs.

Support 5.1 None of Experiment 5.2 GAcSP configurations could satisfy the diameter constraint $c = 2$ or the maximum diameter constraint for 32 - 40 processor PCPs $d_{\text{max}} = 3$. The upper bound on the maximum number of processors for a configuration with diameter $d_{\text{max}} = 3$ is $n_{\text{max}} = 40$, with the Chalmers and Gregory [1992] AMP program achieving a maximum configuration of 32 processors for $d_{\text{max}} = 3$. We can see from Table 5.1 that the GA

component of GAcSP has failed to satisfy the diameter constraint of the 32 processor PCP by 12 paths, each with distance $d_{\text{eval}} = 4$.

We support Claim 5.1 in two ways:

- (1) By suggesting that there are no reasonable grounds for believing, that apart from the 32 PCP, the 33 - 40 node optimum graphs are achievable with $d_{\text{max}} = 3$.
- (2) There is not a significant increase in the mean internode distance due to paths with distance $d_{\text{eval}} = 4$ in increasing processor PCPs.

Optimum graphs are based on the assumption that each node in a graph has the shortest distance possible to all other nodes. That is, every node has a spanning tree configuration in a graph of nodes. This includes the two nodes connected to the system controller (see Section 3.1.2). Increasing PCPs by a factor of one processor substantially increases the work load by adding an extra spanning tree requirement and increasing the spanning tree requirement for all other processors. We stated in Section 3.1.2 that Moore graphs were unattainable for PCPs with $\Delta = 4$. Since our configurations have two processor links used by the system controller, providing less links for connections to reduce the distances, it would seem that obtaining optimum graph configurations is more unlikely than for Moore graphs.

- **Conjecture 5.1** The possibility of achieving theoretical optimum graph configurations with $\Delta = 4$ and 33 - 40 processor PCPs which satisfy $d_{\text{max}} = 3$ decreases with increasing number of processors.

Consider (Figure 5.1) the number of *paths* which do not satisfy $d_{\text{max}} = 3$ for each of the 32 - 40 processor PCPs. We see a monotonic rate of increase (except for the 40 processor PCP). Table 5.2 summarises the results of Equation 3.1 calculations for the percentage reduction

Table 5.2: Summary Experiment 5.2 path calculations									
number processors $k =$	32	33	34	35	36	37	38	39	40
total number <i>paths</i> (k^2)	1024	1089	1156	1225	1296	1369	1444	1521	1600
<i>paths</i>	12	15	24	33	31	47	49	65	62
100% - % d_{avg} decrease	98.83	98.62	97.92	97.31	97.61	96.57	96.61	95.73	96.13

in mean internode distance, due to these extra paths.

total number $paths(k^2)$ - The total number of processor to processor paths for a k processor PCP (i.e. k^2).

100% - % d_{avg} decrease - The decrease in mean internode distance due to the paths which violate $d_{max} = 3$, calculated as a reduction from 100% diameter constraint satisfaction (i.e. $paths/k^2$ see Equation 3.1).

Figure 5.1: Number paths at distance $d_{eval} = 4$ for GAcSP

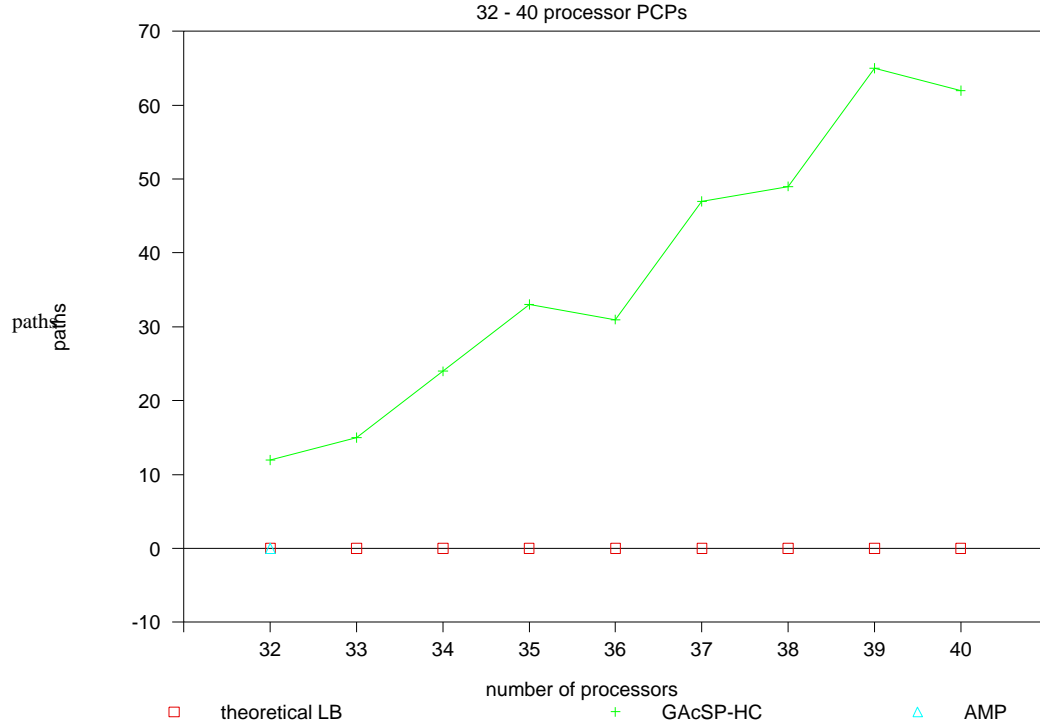
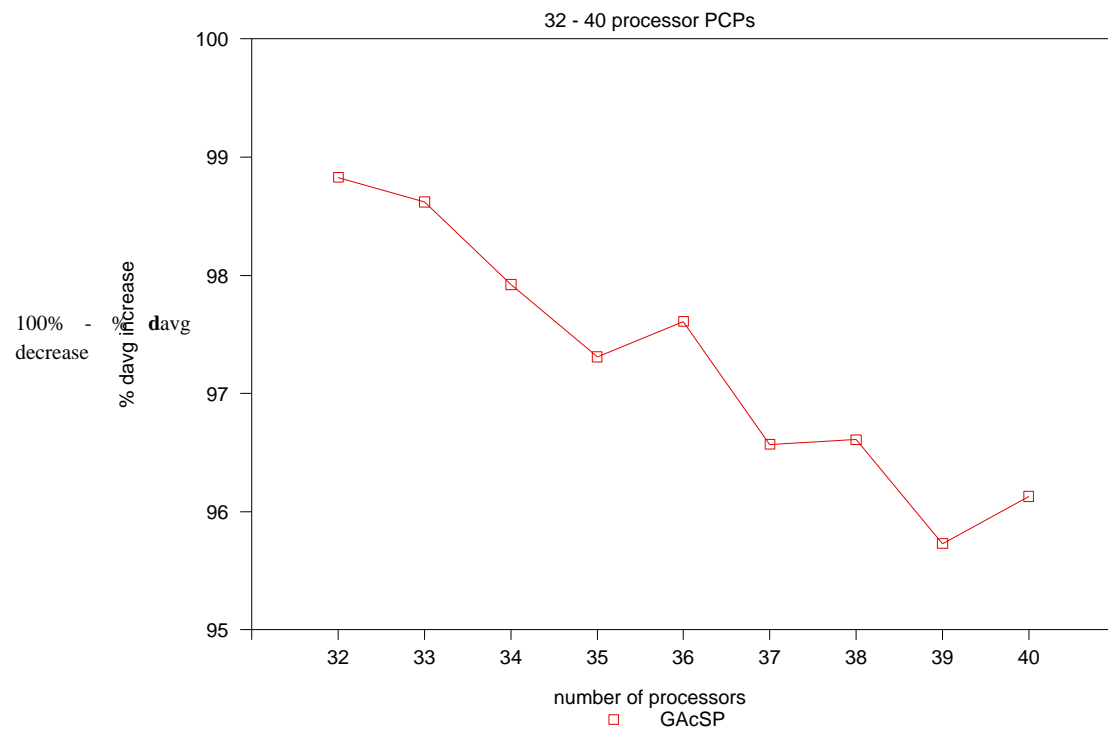
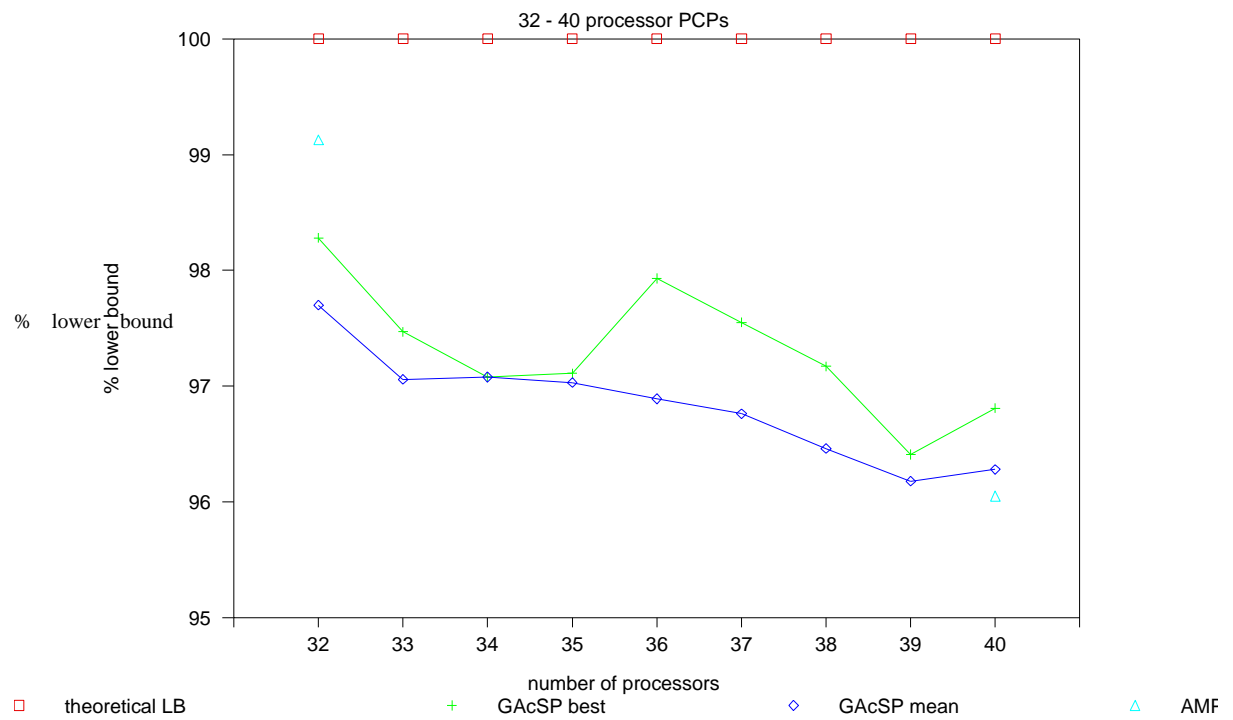


Figure 5.2 shows that the percentage increase in mean internode distance due to the paths at $d_{eval} = 4$ is not significant for the 32 - 40 processor PCPs. Certainly with increasing size of PCPs the worst distance d_{eval} achieved by GAcSP does not exceed $d_{max} = 3$ by more than a factor of 1. We provide further support to Claim 5.1 in Claim 5.2. ■

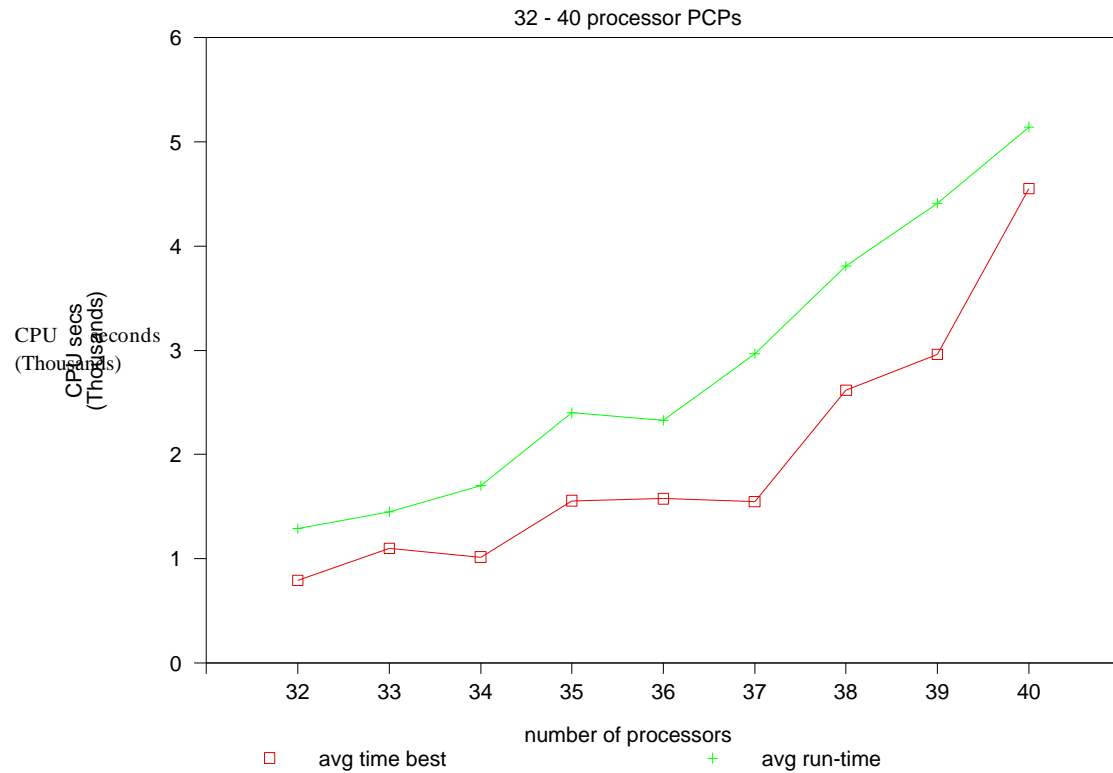
◦ **Claim 5.2** The GA component of GAcSP can provide near optimal mean internode distance results for 32 - 40 processor PCPs in a "reasonable" time period.

Figure 5.2: Percentage decrease in mean internode distance for Experiment 5.2 results**Figure 5.3: Percentage mean internode distances**

Support 5.2 Figure 5.3 illustrates how close to the lower bound D_{avg} solution can be achieved for the 32 - 40 processor PCPs. The graph in Figure 5.3 highlights a comparison on two points (i.e. 32 and 40 processors) between Chalmers and Gregory [1992] PCP program AMP (mean internode distance results) and that of the GAcSP. Although on the 32 processor PCP the AMP is 99.1% of the lower bound whilst that of the GAcSP is 98.3%, on the 40 PCP the GAcSP best of 96.8% is a slight improvement over the 96% for the AMP. It should be noted that the AMP configuration generator is a specially written optimisation program for the PCP [Chalmers and Gregory, 1992], whilst the GAcSP is a generic PCSP solver. There is no evidence to support the idea that the lower bound for the mean internode distance for 32 - 40 processor PCPs is achievable. By setting the diameter constraint $c = 2$ we are effectively making the test PCPs unsolvable. When tackling unsolvable PCSPs GAcSP will always run to convergence unless stopped by a terminating condition (see Section 4.3.2). ■

◦ **Claim 5.3** The time taken to achieve near optimal results for 32 - 40 processor PCPs shows that GAcSP has more potential for solving larger problems than AMP.

Figure 5.4: Average time to achieve best and average run-time in CPU seconds



Support 5.3 The total run-times and the average run-times in CPU seconds (shown in Figure 5.4) taken to obtain the best configurations show that near optimal results for 32 - 40 processor PCPs can be obtained in a acceptable period of time. Furthermore, an exponential amount of time is likely to be required by searching in AMP. On the other hand, GAcSP is not significantly affected by the scaling problem. Therefore, GAcSP has more potential for solving larger problems. ■

5.3 GAcSP Tackling Different Sized PCPs With HC

For *Experiment 5.3* we switched on the GAcSP optional HC (see Figure 4.1) which improves the quality of each offspring after being repaired in the crossover operator. The same nine tests from Experiment 5.2 were carried out for 5 runs each. The GAcSP constant values of all the tests were as Experiment 5.2, including the following:

(e) The maximum time limit for the HC on each offspring generated was 60 CPU seconds.

5.3.1 Results Of Experiment 5.3

All Experiment 5.3 results have been summarised in Table 5.3.

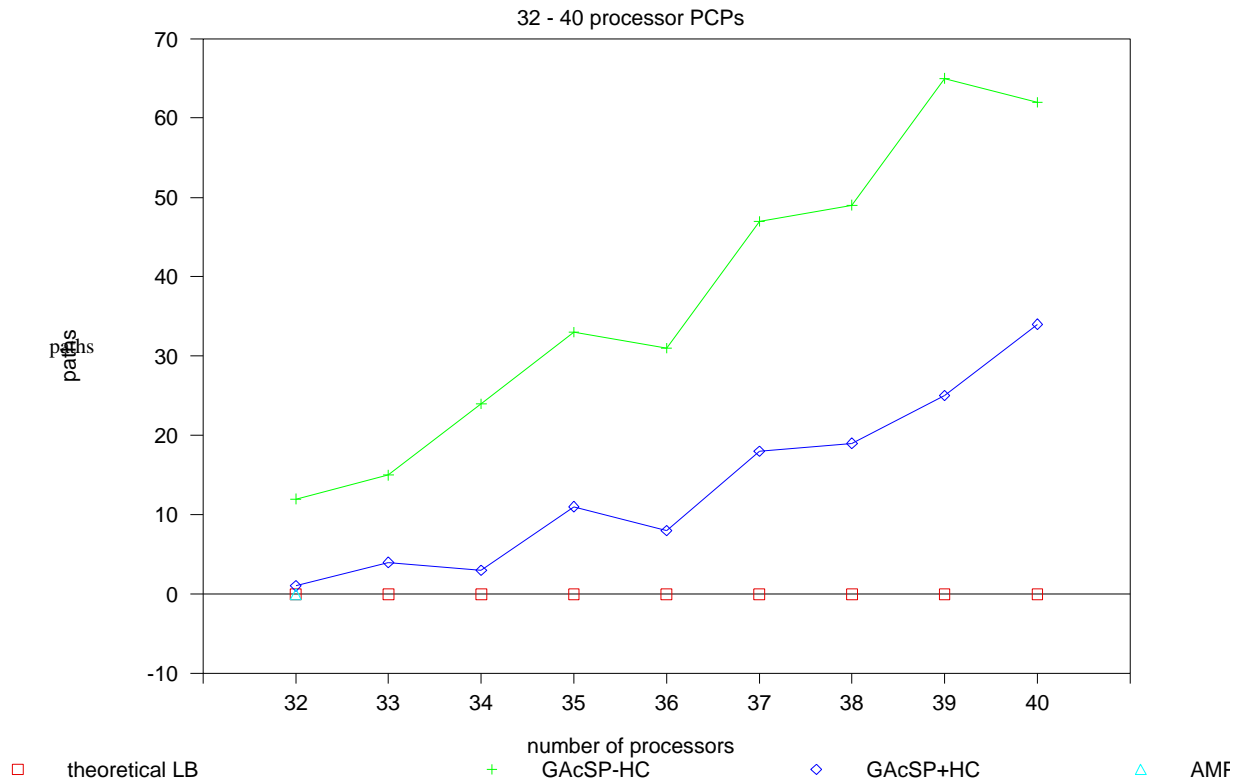
Table 5.3: Summary GAcSP and HC Experiment 5.2 results										
number processors k =	32	33	34	35	36	37	38	39	40	
theoretical LB D_{avg}	2.29	2.31	2.33	2.35	2.37	2.39	2.40	2.42	2.43	
AMP d_{avg}	2.31									2.53
GAcSP+HC Results										
best d_{avg}	2.29	2.32	2.34	2.37	2.38	2.41	2.43	2.45	2.47	
max d_{eval}	4	4	4	4	4	4	4	4	4	
$paths$	1	4	3	11	8	18	19	25	34	
avg d_{avg}	2.298	2.324	2.344	2.372	2.388	2.412	2.430	2.464	2.474	
avg time best sec	9103	4784	5979	6046	10135	7682	10304	11925	14908	
avg run-time sec	8098	8784	8724	10528	12050	11463	17141	13314	18542	

5.3.2 Experiment 5.3 Discussion

- **Claim 5.4** GAcSP can provide near optimal results to 32 - 40 processor PCPs using the HC component for local improvement, which can outperform the specially written program AMP.

Support 5.4 We can see from the results of Table 5.3 and supported by Claims 5.1 and 5.2 that GAcSP can find near optimal solutions but there is room for improvement. In Experiment 5.3 we have switched on HC which gives GAcSP a local improvement ability. In Table 5.3 GAcSP mean internode distance results are much improved and the number of paths not satisfying d_{\max} are also reduced compared with Table 5.1 (see Figures 5.5 and 5.6). The cost of these improvements has been the increased run-times required. These results also show an improvement over the Chalmers and Gregory AMP results, and are close to the lower bound for the mean internode distance. ■

Figure 5.5: Number of paths $d_{\text{eval}} = 4$ achieved by GAcSP



5.4 Summary

We have tested GAcSP on the processors configuration problem (PCP), for a group of PCPs with 32 to 40 processors. The task of PCP is to configure the processors with the objective of minimising the mean internode distance. The first tests were carried out on the group of PCPs using GAcSP without the hill-climbing (HC) component. The second tests on the group of PCPs using GAcSP with the HC component switched on. The test conditions were a GAcSP population size of 80 strings, diameter constraint $c = 2$, 5 runs for each test, and the HC component had a 60 CPU second maximum time limit. The "best" and average (minimum) mean internode distance results along with the time taken to terminate for these tests were summarised and presented in table form. These results were analysed and used, along with graphs and statistical analysis to support the following claims:

- The GA component of GAcSP can provide consistent near optimal diameter constraint satisfaction results to 32 - 40 processor PCPs. Which we support by suggesting that there are no reasonable grounds for believing, that apart from the 32 PCP, the 33 - 40 node optimum graphs are achievable with $d_{\max} = 3$. There is not a significant increase in the mean internode distance due to paths with distance $d_{\text{eval}} = 4$ in increasing processor PCPs.
- The GA component of GAcSP can provide near optimal mean internode distance results for 32 - 40 processor PCPs in a "reasonable" time period. Supported by considering that the diameter constraint as $c = 2$ setting effectively makes the test PCPs unsolvable. When tackling unsolvable PCSPs GAcSP will always run to convergence unless stopped by a terminating condition.
- The time taken to achieve near optimal results for 32 - 40 processor PCPs shows that GAcSP has more potential for solving larger problems than AMP. Supported by the average CPU second run-times and the consistent near optimal results achieved on increasing sized PCPs.

- GAcSP can provide near optimal results to 32 - 40 processor PCPs using the HC component for local improvement, which can outperform the specially written program AMP. Switching on HC improves mean internode distance results and show an improvement over the Chalmers and Gregory [1992] AMP results, and are close to the lower bound for the mean internode distance.

Chapter 6 CarSP Results

6.1 Plan

The purpose of this chapter is to report on the results for GAcSP in tackling the CarSP as our second example PCSP. The main issues we wish to test in this chapter concerns GAcSP's ability to cope with loose and tight constraints, problems of increasing size, and the performance of GAcSP on unsolvable problems. CarSPs with different characteristics are considered in the following sections:

- 6.2 GAcSP Tackling CarSPs Of Different Tightness
- 6.3 GAcSP Tackling CarSPs Of Different Size
- 6.4 GAcSP Tackling Unsolvable CarSPs

In each section we describe the different characteristic CarSPs tested. These CarSPs provide a measure of GAcSP performance and allow a comparison with available published research results. We can also compare our results with those from fellow researchers working on CarSPs in the GENET project at Essex University. GENET is a connectionist model for CSP solving [Wang and Tsang, 1991]. CarSPs were generated by a program (supplied by Kangmin Zhu) which provided a solution to each problem. Kangmin Zhu is a researcher working on the GENET project at Essex University.

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 directly compare our results with those of other researchers ([Dincbas et al., 1986; Parrello, 1988; Parrello et al., 1986]). CarSPs with these capacity constraints represent two important aspects of CarSPs in general. Firstly they reflect the characteristics of real-world CarSPs. Secondly they give rise to a range of "difficulty" in utility ratios (defined and analysed in a later section). In real-world CarSPs there are practical constraints of time, assembly line space, workstation space, and the constant speed of the assembly line. The need to efficiently manage these practical constraints may limit the workstation team size. The utility ratios need to be considered as part of a larger system, which could include separating jobs

into smaller units and work undertaken off the assembly line. For example, the number of cars which pass through a workstation representing the time it takes a team to fit an option, and the speed of the assembly line will dictate the workstation space required. The $1/2$, $2/3$, $1/3$, and $1/5$ capacity constraints used in the test CarSPs reflect the objective of minimising production costs by utilising workstation space and teams. The utility ratios represent a range of capacity constraints from, one in five cars ($1/5$) to two cars in three ($2/3$) requiring options. In our experiments we define a measure of CarSP difficulty and evaluate the algorithms performance in tackling them. By considering both the range of utility ratios and the interaction between the capacity constraints we hope to make limited generalisations about CarSPs in general.

For all tests undertaken, GAcSP constants are described with their values, these include population size, number of lowest fitness members copied directly into the mating pool each cycle, number of offspring created each cycle, maximum number of cycles and run-time. The HC is switched on for all tests undertaken and the maximum time limit for improvement of each offspring is given. We summarise and present the results from the tests in tabular form with explanations for the terms used. Claims are made and *followed* by supporting analysis of the results, assisted by graphs and analysis of variance *F*-test statistics.

6.2 GAcSP Tackling CarSPs Of Different Tightness

In *Experiment 6.2*, tests were undertaken on solvable CarSPs with average utility \hat{u} from .45 to .90 in intervals of .05. All CarSPs were generated by a program (supplied by Kangmin Zhu) which provided a solution to each problem, where all capacity constraints were satisfied. The number of car types k in all CarSPs generated, is variable between $1 \leq k \leq 2^n$. For each of the 10 average utilities 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 average utility test.

As well as GAcSP with the optional HC switched on, a *heuristic repair* algorithm (HR) and a heuristic repair combined with a single state *TABU* (TABU) search algorithm were tested on these problems. The heuristic repair method uses repair choices with minimum constraint conflicts [Minton et al., 1990] to reduce the violated constraints. TABU search is a generic

search strategy for optimisation problems [Glover, 1989; 1990]. The HR and TABU results were supplied by Kangmin Zhu along with the pseudo code listed in Appendix A, for these programs. In order to compare results between the HR and TABU algorithms the CarSP option weighted penalty values used in GAcSP tests were all equal to 1, i.e. $\forall(p_{mr}) = 1$ for $r \geq p_m = 0$. The objective function for the algorithms calculated the number of options violated in a schedule. All the algorithm tests were run on a SUN 4/110 under UNIX 4.0 operating system. The GAcSP constant values of all the tests were:

- (a) A population of 80 strings were randomly generated each test run [Tsang and Warwick, 1989].
- (b) 10% of the minimal fitness (elite) population strings were copied directly into the mating pool at reproduction phase of GAcSP.
- (c) The number of offspring created each GAcSP cycle was arbitrarily set at four.
- (d) GAcSP termination conditions for each test run were set at maximum generations = 400, maximum run-time = 10 CPU hours.
- (e) The maximum time limit for HC was 30 CPU seconds for each offspring.

6.2.1 Results Of Experiment 6.2

GAcSP, HR and TABU results from Experiment 6.2 have been summarised in Table 6.1.

Table 6.1 explanation.

avg utility \hat{u}	- Average utilities from .45 - .90 (see Equation 3.12) in steps of .05.
avg car types k	- The mean of the 10 car types for each CarSP average utility.
number solns	- The number of runs returning solutions for each average utility.
min violation	- The best (minimum) solution for each average utility. (Only given for TABU; see below for explanation.)

avg violation - The mean of all test run minimum violations (including solutions) for each average utility.

avg run-time *sec* - The mean of all the run-times (GAcSP termination) in CPU seconds taken for each average utility.

Table 6.1: Summary GAcSP, HR and TABU Experiment 6.2 results										
avg utility \hat{u}	.45	.50	.55	.60	.65	.70	.75	.80	.85	.90
avg car types k	8.7	12.3	12.9	16	18.2	20	21.2	22	23.4	23.3
GAcSP Results										
number solns	100	100	99	100	99	99	92	61	21	1
avg violation	0.00	0.00	0.01	0.00	0.01	0.01	0.09	0.50	1.40	3.90
avg run-time <i>sec</i>	29	49	69	43	60	212	457	1122	2104	4421
HR Results										
number solns	98	97	94	96	94	97	88	58	15	1
avg violation	0.02	0.04	0.07	0.08	0.07	0.04	0.19	1.04	2.42	7.00
avg run-time <i>sec</i>	19	26	46	40	57	44	144	451	856	975
TABU Results										
number solns	100	100	100	100	100	100	97	21	0	0
min violation	0	0	0	0	0	0	0	0	2	6
avg violation	0.00	0.00	0.00	0.00	0.00	0.00	0.05	1.62	5.74	11.85
avg run-time <i>sec</i>	4	6	11	4	8	10	111	818	956	960

6.2.2 Experiment 6.2 Discussion

Table 6.1 summarises the results from testing GAcSP, HR and TABU on solvable CarSPs. The number of solutions indicates how many of the individual (average utility) test runs returned a solution satisfying the capacity constraints. Because algorithm TABU failed to find any solutions in the .85 and .90 average utility tests, minimum violation results are presented. When no solutions are found, the minimum violation is the string with the least number of options violated. The mean of all minimum violations is given, and mean run times for each test.

- ° **Claim 6.1** GAcSP can find more solutions to increasing .45 to .90 average utility CarSPs than the HR algorithm.

Support 6.1 There is a statistically significant difference between the number of solutions found by GAcSP and HR (see Table 6.2). Figure 6.1 demonstrates that GAcSP finds more solutions than HR for all average utility tightness Experiment 6.2 CarSPs. ■

Table 6.2: Summary Experiment 6.2 result F -test statistics - significance in parenthesis

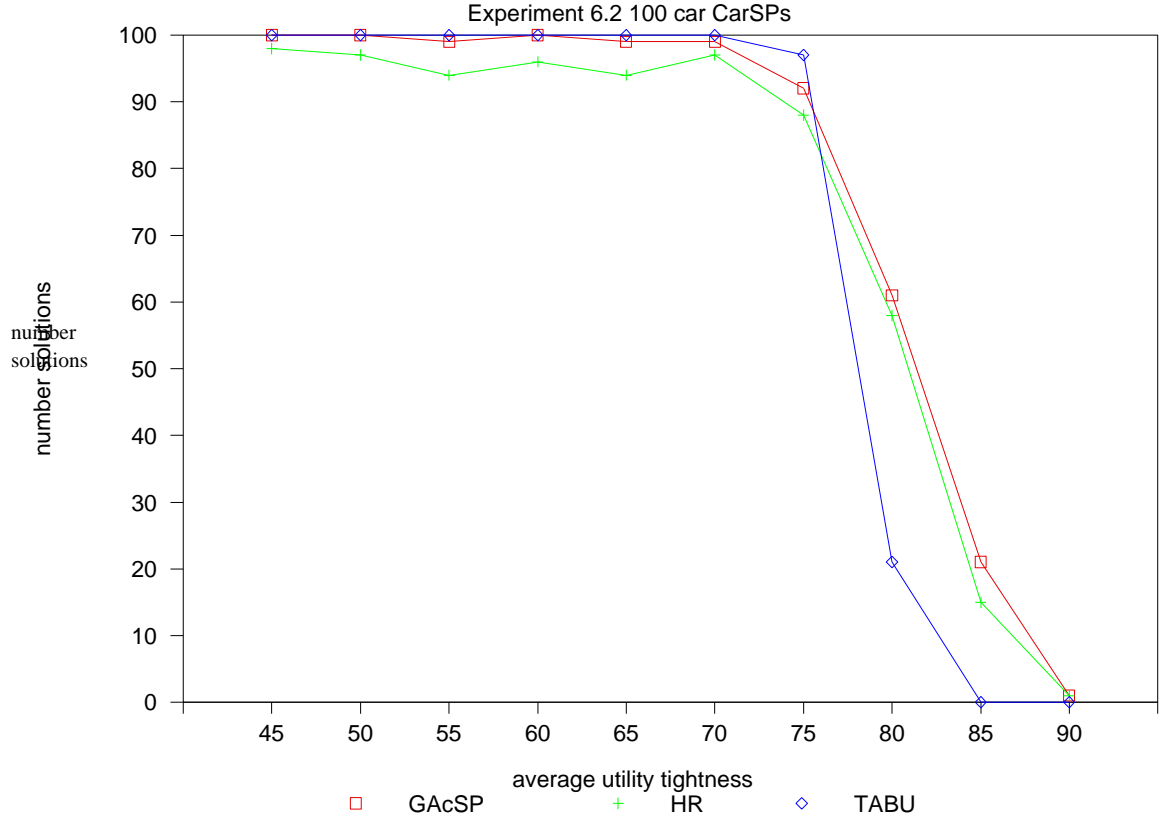
hypothesis	Observed F value			Criterion F value	
	number	solns	avg violation	avg run-time	
compare algorithms					
GAcSP with HR	(36.63)	2.73	2.99	10.6	5.12

Observed F values are calculated from the sum of squares of the standard deviations. The criterion F value at levels $\alpha = 0.01$ and $\alpha = 0.05$ are recorded from a statistical table of the F distribution, indexed by the degrees of freedom. Where the observed F value for a particular performance measure comparison is greater than the criterion F value, there is a statistically significant difference between the data groups with a stronger degree of confidence at the $\alpha = 0.01$ level than $\alpha = 0.05$.

The analysis of variance F -test statistics are summarised in Table 6.2. In Table 6.2 we have statistically compared the performance results from Table 6.1, between the algorithms tested. Our significant statistical conclusion from Table 6.2 is: (a) There is a 99% level of confidence to reject the hypothesis that there is no correlation between the performance of GAcSP and HR in finding solutions to Experiment 6.2 CarSPs.

- ° **Claim 6.2** The GAcSP strategy has achieved better average performance in finding solutions to average utility .45 - .90 tests than HR and TABU.

Support 6.2 GAcSP has found solutions for all runs in the .45, .50, and .60 average utility tests and found 99 solutions in runs for .55, .65 and .70 average utility tests. GAcSP average performance for finding solutions in .45 - .70 tests is 99.5 runs. TABU has found solutions

Figure 6.1: GAcSP, HR and TABU Experiment 6.2 solutions comparison

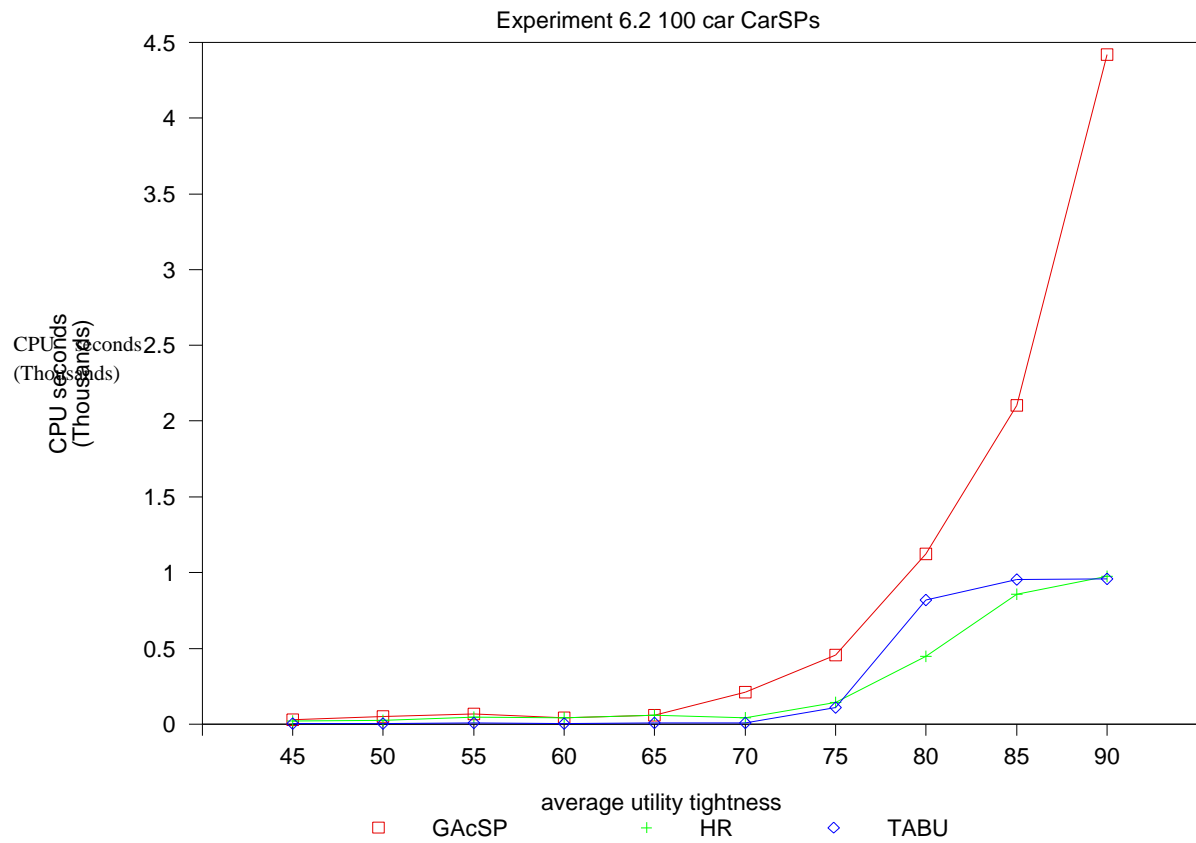
for all runs in the .45 - .70 utility tests. If we calculate the average number of runs finding solutions by each algorithm for all Experiment 6.2 tests, presented in Table 6.1 GAcSP has an average 77.2, HR 73.8 and TABU 71.8. ■

° **Claim 6.3** The GAcSP performance (*robust*) in finding solutions to complex tightly constrained average utility CarSPs is better than HR and TABU because GAcSP does not depend entirely upon local search information.

Support 6.3 As the average utility is increased the complexity of CarSPs is increased. Below we explain the source of this complexity and suggest how it effects the performance of each algorithm. Also, why the performance of GAcSP in finding solutions and minimising violations is not as reduced as HR and TABU when put under the pressure of increasingly complex CarSPs.

Both GAcSP and TABU are able to find solutions in nearly all .45 - .75 average utility runs within reasonably quick run-times (see Figure 6.2). However, it is only at the .80 - .90 average

Figure 6.2: GAcSP average CPU run-times for Experiment 6.2 CarSPs



utility results that we can clearly distinguish between the behaviours 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 - .90, with GAcSP finding more solutions than HR. This dramatic reduction in performance of the algorithms on the .80 average utility test is probably due to differences in CarSP characteristics leading to increased tightness. A number of problem characteristics are held constant for all test CarSPs. These include number of cars (100), number of options (5), and capacity constraints (1/2, 2/3, 1/3, 2/5 and 1/5). In our test CarSPs the difference between average utilities is due to a combination of the number of car types, car types used and the production requirements.

We can note from Table 6.1 that there is only a difference of .8 in the average number of car types between the average utility of .75 and .80. Table 6.3 shows the search space sizes

Table 6.3: Search space sizes for $N=100$ car Experiment 6.2 CarSPs					
avg utility \hat{u}	.45	.50	.55	.60	.65
avg car types k	8.7	12.3	12.9	16	18.2
search space size N^k	2.5E+17	4.0E+24	6.3E+25	1.0E+32	2.5E+36
avg utility \hat{u}	.70	.75	.80	.85	.90
avg car types k	20	21.2	22	23.4	23.3
search space size N^k	1.0E+40	2.5E+42	1.0E+44	6.3E+46	4.0E+46

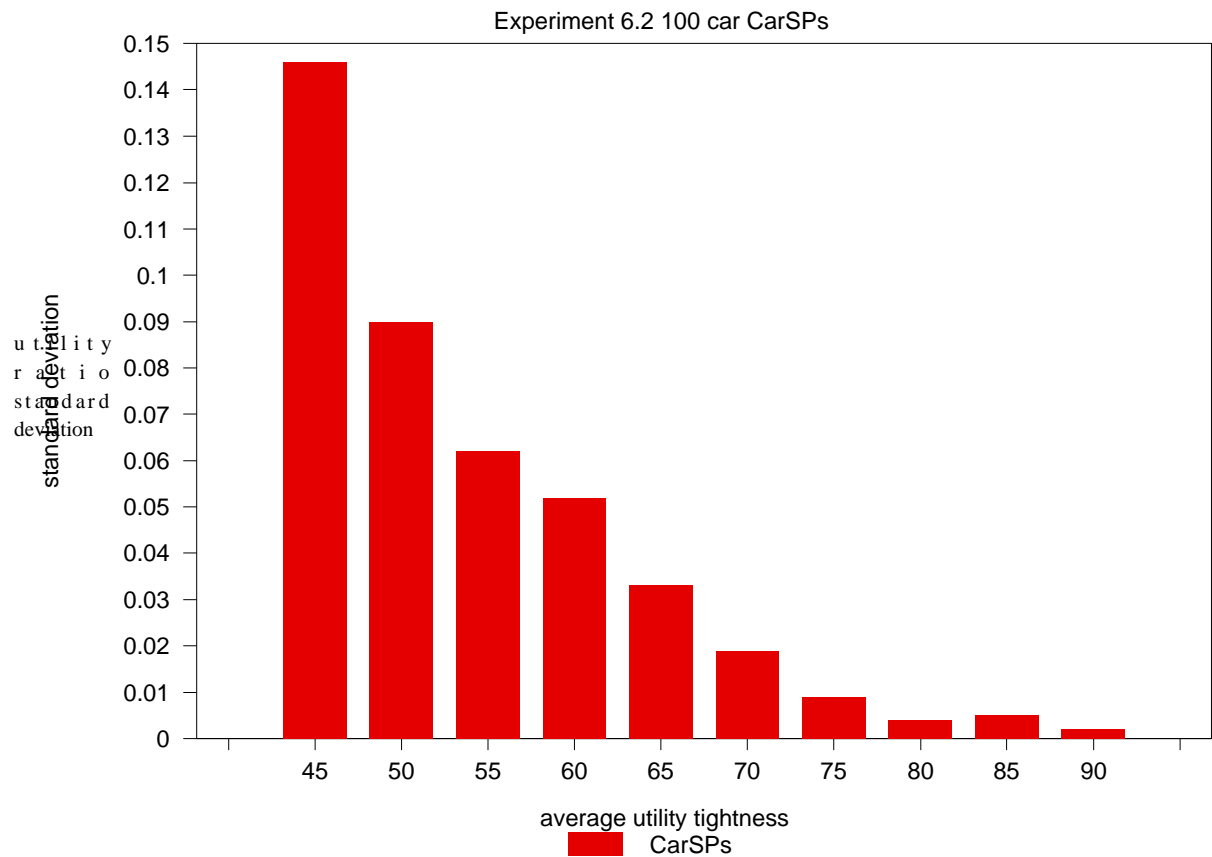
for Experiment 6.2 CarSPs and provides information for analysing the difference in solution finding results for the algorithms. The number of car types in a CarSP makes a significant difference in the search space size but does not necessarily reflect the difficulty faced by the algorithms in finding solutions. In fact, the greatest difference of 3.6 between .45 and .50 makes no difference to the number of solutions found. Although the ratio of solutions to search space size decreases for .45 - .70 tests, the results in Table 6.1 show that this factor alone does not deter any of the algorithms from finding solutions. If we look more closely at the ten individual test CarSPs results in the .75 and .80 average utility in Table 6.4, we find that the

Table 6.4: .75 and .80 CarSPs test run results											
test number	1	2	3	4	5	6	7	8	9	10	avg/tot
.75 car types $k=$	20	23	19	23	21	23	20	21	23	19	21.2
.75 number solns	8	10	10	10	10	10	10	8	6	10	92
.80 car types $k=$	21	24	22	22	21	23	23	22	20	22	22
.80 number solns	3	9	10	3	8	1	5	6	10	6	61

number of solutions found by GAcSP does not correlate with the number of car types. For example, GAcSP has found solutions in all runs (i.e. 10) for the .75 average utility test 6 with $k = 23$ car types, whilst only one run returned a solution in the .80 average utility test 6 CarSP with the same number of car types. The number of car types in a CarSP will depend upon the utility ratio for each option. The number of average car types in Table 6.1 increases

in order to generate tighter CarSPs. Figure 6.3 shows the standard deviation for the spread of utility ratios for Experiment 6.2 CarSPs which reduces as the average utility increases. Given these standard deviations it would be unlikely that the loss of performance at .80 is due to a substantial imbalance in utility ratios. The car types which require less options make the task

Figure 6.3: Standard deviations for Experiment 6.2 CarSPs utility ratios



of generating a solvable CarSP easier. As increased average utilities are required, car types which are more difficult to place, need to be used. These more difficult car types are those which require the most options. It is the interactions between the options that make CarSPs difficult to solve for the algorithms tested. This option interaction is due to a combination of the number of options in the car types and the capacity constraints. Parrello [1988] recognised and used this car type difficulty as a way to discriminate and sequence such car types before others.

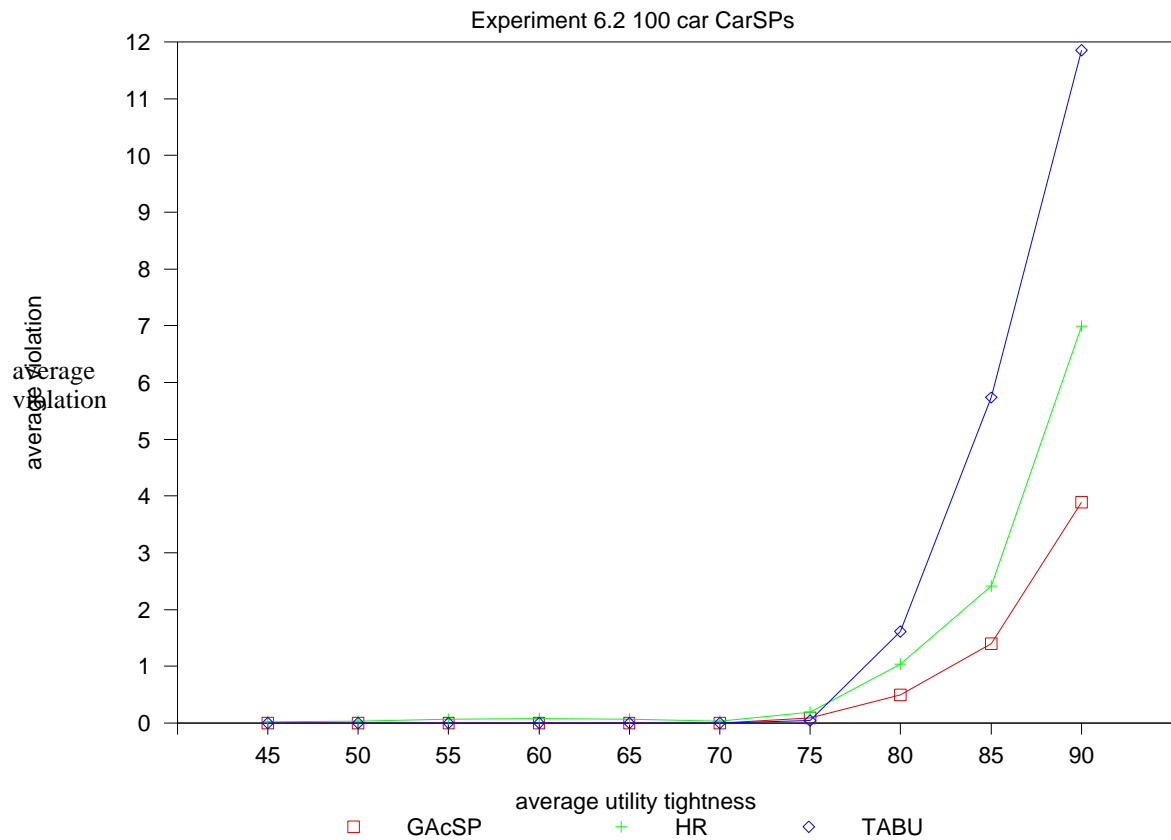
HR and TABU are more dependent for exploration upon local search space information at a single point than GAcSP which uses a population of points. With increasing option interactions local information is less complete in guiding the search towards solutions and the starting points used are more important. One cause of reduced local information occurs when a car in a schedule can violate more than one option. This can create two difficulties for local search techniques including fewer alternative positions to move cars and reduce their option violations, and more cars with option violations. Both HR and TABU will suffer from their dependence on the quality of good starting points, with TABU expected to perform better than HR. TABU search performs better because it has the ability to escape from local minima, by using a form of short term memory for previous choices. By increasing the tightness of CarSPs the fitness space may provide reduced feedback to guide both the GA and HC components of GAcSP search. Although both GAcSP components suffer from a lack of complete information, GA uses a population of points which prevents it from becoming easily trapped in local minima. With increasing tightness of CarSPs the HC will contribute less directly to the search process but will still assist in the development of good building blocks. In this case the work of the GA component is increased. This balance of work between the GA and HC components is an important feature of GAcSP and ensures a robust approach. Changing the maximum time allowed for HC will alter the balance between exploitation and exploration. This will depend upon the requirements of the GAcSP *user* who accepts that time saved in exploration will result in less rigorous search. Furthermore, GAcSP allows the opportunity to fine tune parameters which could improve performance. But because HR and TABU are more dependent upon local information it may not be possible to improve their performance beyond the results already reported. ■

- ° **Claim 6.4** GAcSP can provide more consistent near optimal performance on average minimal violation results for increasing average utility tightness in Experiment 6.2 CarSPs than HR and TABU.

Support 6.4 In the .45 - .75 tightness tests the average minimum violation results are dominated by the number of solutions found by the algorithms. But as GAcSP, HR and TABU find less solutions in the .80 - .90 tightness 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 tightness tests above .75 significantly decline, the reductions in the quality of average minimum violation results are not as severe. Figure 6.4 shows that the rate of increase in GAcSP average violation results for tightness .80 - .90 is not as great as for HR and TABU. GAcSP can retain a near optimal performance even at .90 tightness. (In

Figure 6.4: GAcSP, HR and TABU average violation Experiment 6.2 results



Experiment 6.4 we demonstrate that GAcSP can retain this near optimal performance on tightness greater than .90.) The mean of average minimal violation results of GAcSP, HR and TABU on Experiment 6.2 CarSPs are GAcSP 0.592, HR 1.097 and TABU 1.926. Claim 6.5 further supports Claim 6.4, demonstrating the robustness of GAcSP. ■

° **Claim 6.5** GAcSP has the opportunity to improve the quality of its performance on increasing average utility CarSPs which is not reduced to the extent of that shown by HR and TABU.

Support 6.5 As the average utility tightness is increased we would expect GAcSP or any algorithm to work harder in finding solutions. This is evident, particularly at a critical .80 utility tightness. Given Definition 6.1, as we increase the tightness of CarSPs the ratio of search space solutions to the size of search space will decrease. GAcSP will need to work harder to find solutions as CarSP utility tightness increases. The extra work undertaken by GAcSP can be seen in increasing run-times in Figure 6.2. Since the only difference between Experiment 6.2 CarSPs is the increasing utility tightness, the extra CPU time required by GAcSP is due to more intensive evaluations. More intensive evaluations are required because of increasing numbers of car types and the number of options in each car type. These extra evaluations slow GAcSP down but do not prevent it from maintaining consistent near optimal results. We have the opportunity with GAcSP, by increasing the maximum improvement time for HC (e.g. from 30 CPU seconds to 40 CPU seconds), to improve the performance of GAcSP. However, if we allow more time to HR and TABU the performance is unlikely to be improved. Because HR and TABU work from a single search space point they are more likely to settle in local minima.

There is a correlation between the run-times shown in Figure 6.3 for average utilities .45 - .75 and the number of solutions found. All algorithms terminate when finding a solution, and therefore the run-times for .45 - .75 utility tightness represent the time taken to find solutions.

■

6.3 GAcSP Tackling CarSPs Of Different Size

Experiment 6.3 tests were undertaken on solvable CarSPs with 100, 120, 140, 160, 180 and 200 cars to sequence, and average utility ratios .50, .60, .70, and .80. There were 5 randomly generated CarSPs for each average utility ratio and 5 runs carried out on each problem. All CarSPs were generated by a program (supplied by Kangmin Zhu) which provided a solution to each problem. The GAcSP constant values of all the tests were as in Experiment 6.2.

6.3.1 Results Of Experiment 6.3

All results from the tests have been summarised in Tables 6.5 and 6.6.

Table 6.5:	(a) .50 average utility ratio						(b) .60 average utility ratio					
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

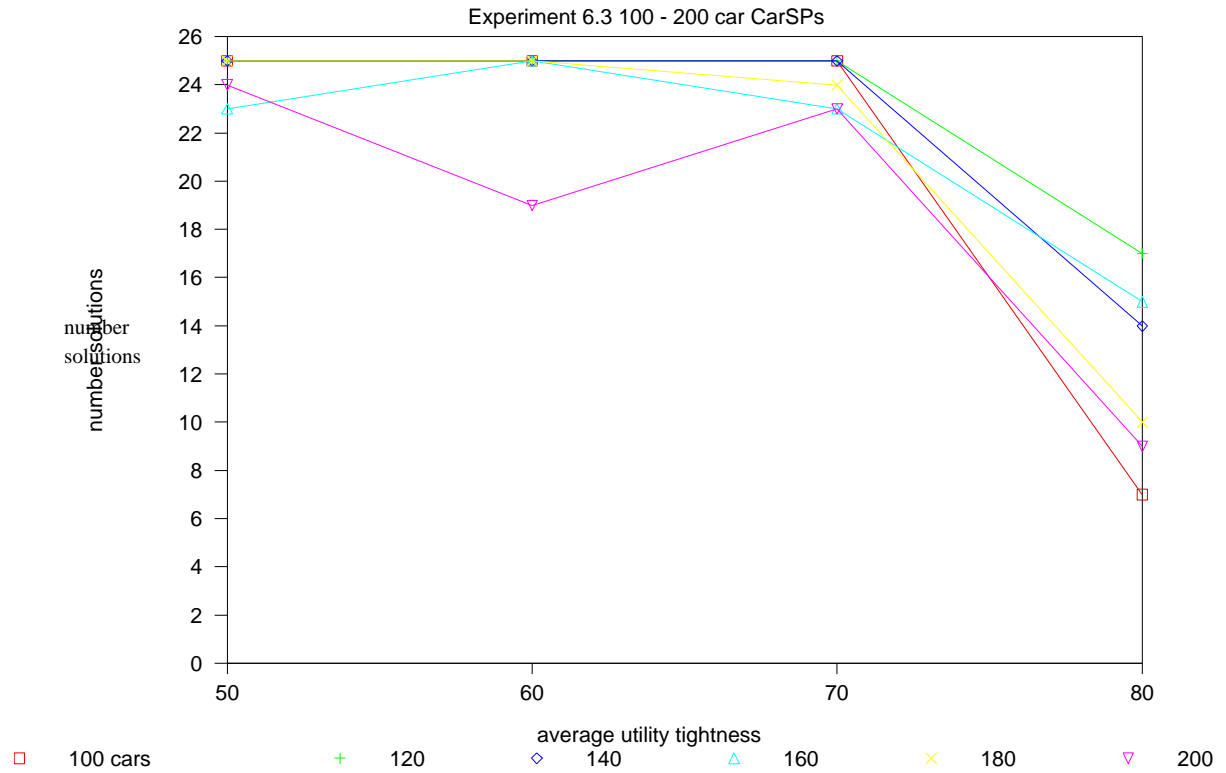
Table 6.6:	(a) .70 average utility ratio						(b) .80 average utility ratio					
number cars N	100	120	140	160	180	200	100	120	140	160	180	200
avg car types k	20.6	21.4	22.8	21.6	25	25	22.6	22.4	24.2	24.2	25.8	26.2
number solns	25	25	25	23	24	23	7	17	14	15	10	9
avg violation	0	0	0	.08	.04	0.12	.92	.32	.92	1.36	.88	1.2
avg run-time sec	339	369	699	1704	539	4177	2033	3422	5261	7079	5969	7289

6.3.2 Experiment 6.3 Discussion

Tables 6.5 and 6.6 summarise the results from testing GAcSP on solvable 100, 120, 140, 160, 180, and 200 CarSPs with average utility .50, .60, .70 and .80. The same GAcSP performance statistics were summarised in Tables 6.5 and 6.6 as in Table 6.1.

° **Claim 6.6** The performance of GAcSP in finding solutions to CarSPs is not significantly reduced by problem size.

Support 6.6 The solution finding performance of GAcSP decreases as the problems become tighter, supporting our observations from Experiment 6.2. This trend can be seen clearly in Figure 6.5. Although we can see from Figure 6.5 that there is a slight reduction in the number of solutions with the increase in the number of cars, generally the performance of GAcSP is consistent. The loss in performance is not significant (see Table 6.7), yet the increase in search space size for these CarSPs is significant (see Table 6.8). 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 behaviour is an important design feature and suggests that the time allowed to HC depends more on problem characteristics of the number of car

Figure 6.5: The number solutions found by GAcSP for Experiment 6.3 CarSPs

type options and production requirements, as we have seen with Experiment 6.2 tests than problem size. This emphasises the fact that the GA component of GAcSP ensures robustness (Claim 6.5) whilst the HC component adds a specialist ability.

Table 6.7: Summary Experiment 6.3 result F -test statistics - significance in parenthesis

hypothesis	Observed F value				Criterion F value	
	number solns	avg violation	avg schedule	avg run-time	$\alpha=0.01$	$\alpha= 0.05$
number cars k	1.87	(5.72)	2.21	2.79	4.56	2.90

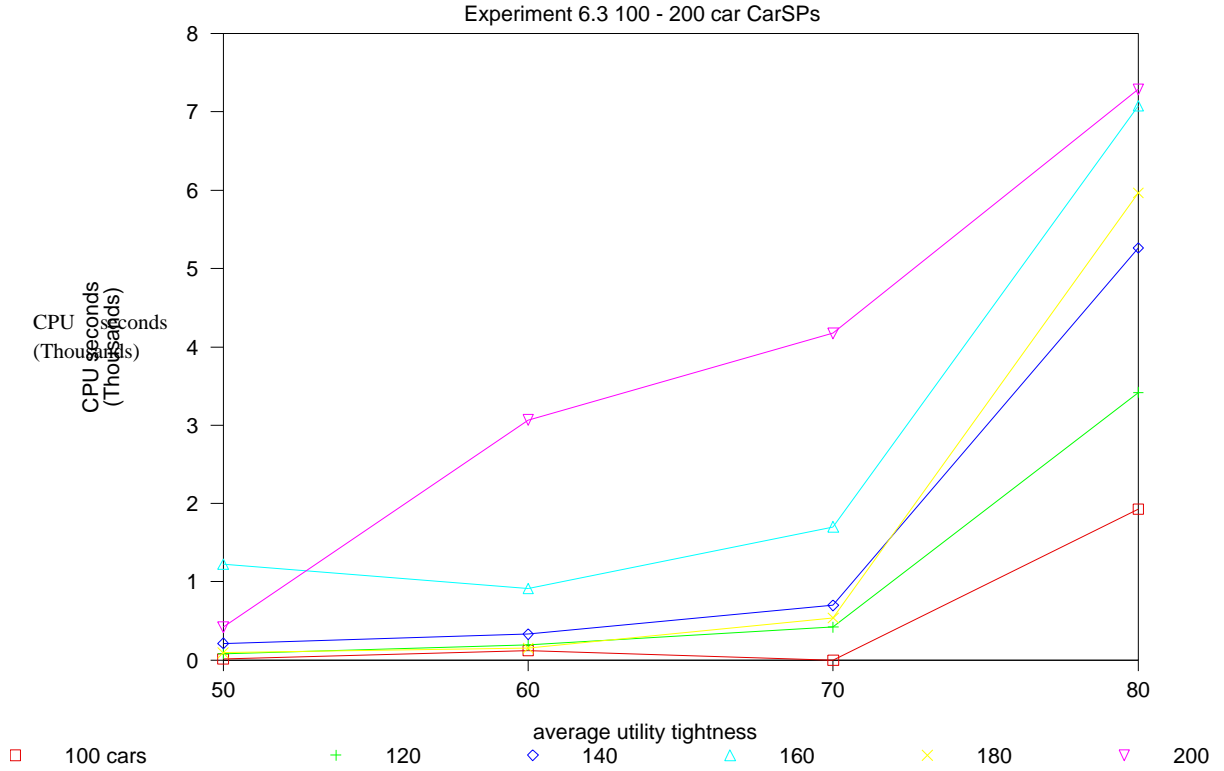
The analysis of variance F -test statistics are summarised in Table 6.7. In Table 6.7 we have statistically compared the performance results from Tables 6.5 and 6.6, between the number of cars. Our significant statistical conclusions from Table 6.7 are: (a) There is a 99% level of confidence to reject the hypothesis that there is no correlation between the number of cars and the average violation achieved by GAcSP for Experiment 6.3 CarSPs.

Table 6.8: Search space sizes for Experiment 6.3 CarSPs

types - size	k	k	k	k	N^k	N^k	N^k	N^k
avg utility \hat{u}	.50	.60	.70	.80	.50	.60	.70	.80
$N = 100$	12.6	16.4	20.6	22.6	1.6E+25	6.3E+32	1.6E+41	1.6E+45
$N = 120$	11.2	17.4	21.4	22.4	1.9E+23	1.5E+36	3.1E+44	3.7E+46
$N = 140$	12.2	18.2	22.8	24.2	1.5E+26	1.1E+39	8.5E+48	8.6E+51
$N = 160$	11.4	19.6	21.6	24.2	1.3E+25	1.6E+43	4.1E+47	2.2E+53
$N = 180$	17.8	22.6	25.0	25.8	1.4E+40	9.3E+50	2.4E+56	1.5E+58
$N = 200$	15.2	22.2	25.0	26.2	9.5E+34	1.2E+51	3.4E+57	1.9E+60

In general the ability of GA to find solutions is not necessarily restricted by search space size. One important effect of increasing the CarSP size is to increase the computational workload of the GA which can slow the GA down. This increase in the case of the GAcSP is due mainly to the CPU requirements of the evaluation function and crossover mechanism. The average CPU second run-time in Figure 6.6 shows this increase for all run-time averages shown in Tables 6.5 and 6.6 with the exception of the 180 CarSP, with .50 average utility. In this case, all the test runs resulted in solutions enabling the GAcSP to terminate before complete convergence. Figure 6.6 demonstrates this difference between the average utility tests where GAcSP terminates on finding solutions and those which are run to convergence. ■

We can make a limited comparison of the results from Experiments 6.2 and 6.3, with those reported by Parrello in [1988] and Parrello *et al.* in [1986]. Parrello *et al.* used an Automated Reasoning Program they called ITP to sequence 5 cars with 5 options. This took 35 minutes, and 15 minutes with ITP written using OPS5. Dincbas *et al.* [1988] have developed a Constraint Logic Programming Language (CHIP) which tackled solvable CarSPs. They reported that CHIP could sequence 100 car schedules with an average utilisation of .80 in under 60 seconds and 200 cars between 336 and 345 seconds. Although their approach showed good results it was restricted to solvable CarSPs only.

Figure 6.6: GAcSP average CPU second run-times for Experiment 6.3

6.4 GAcSP Tackling Unsolvable CarSPs

Experiment 6.4 tests were undertaken on unsolvable CarSPs, each with a single over-utilised option. The goal of *Experiment 6.4* is to test the effect on GAcSP of tackling CarSPs with over-utilised utility ratios. We carried out tests on 4 groups of problems. The groups named .50, .60, .70, and .80 are based on the average utility *Experiment 6.2* CarSPs used to generate the unsolvable CarSPs. Each group has 25 unsolvable CarSPs derived from *Experiment 6.2* CarSPs.

From each CarSP in *Experiment 6.2*, we produced 5 new unsolvable CarSPs by over-utilising each of the 1 - 5 options. The group of 5 unsolvable CarSPs derived from each *Experiment 6.2* CarSP are called problem 1 to problem 5. For each option m , where $m = 1, 2, \dots, 5$ the solvable CarSP has option m added to randomly selected car types until u_m is over-utilised

(i.e. $u_m > 1$). For example, to obtain the first Experiment 6.4 test .50 unsolvable CarSP we take the Experiment 6.2 .50 solvable CarSP and add option 1 to randomly selected car types which do not already have option 1, until option 1 is over- utilised. We do the same for option 2 using the solvable .50 CarSP to get the second unsolvable CarSP, and so on for each option resulting in 5 unsolvable CarSPs. There were 10 runs for each unsolvable CarSP and therefore a total of 50 runs for each option and average utility. The GAcSP constant values of all the tests were as Experiment 6.2.

6.4.1 Results Of Experiment 6.4

GAcSP results from Experiment 6.4 have been summarised in Tables 6.9, 6.10 and Figure 6.7.

Table 6.9: Number of minimum violations above minimum lower bound										
violation from LB	0	1	2	3	5	6	7	8	13	15
.50 number violations	6	8	1	3		4		1	1	1
.50 % violations	24	32	4	12		16		4	4	4
.60 number violations	8	10	5		1		1			
.60 % violations	32	40	20		4		4			
.70 number violations	7	9	5	4						
.70 % violations	28	36	20	16						
.80 number violations	4	7	5	7	1		1			
.80 % violations	16	28	20	28	4		4			

Table 6.9 explanation.

violations from LB - The number of violations from the theoretical LB.

number violations - The number of GAcSP minimal violation tests from the theoretical lower bound.

% violations - The number of GAcSP minimal violation tests from the theoretical LB expressed as a percentage of the maximum possible.

Table 6.9 summarises the number of GAcSP violation results (rows) achieved at each violation *distance* (columns) from the theoretical LB. Entries in the first column of Table 6.9 (e.g. 0 violation from LB) represents how many of each group tests have achieved the theoretical LB. Since there are 25 unsolvable CarSP tests in each group, the maximum number which can achieve the theoretical LB is 25. The percentage is calculated as the number achieved over the maximum possible. The second column to the last in Table 6.9 represent increasing number of violations from the theoretical LB. For example in group .50, one of the 25 tests at column 10 returned a minimal violation which was 15 violations from the theoretical LB. For violations from LB not indicated in Table 6.9 (e.g. 4, 9) there were no tests for any group with these violations from the theoretical LB.

Table 6.10: Summary Experiment 6.4 CarSPs					
	problem				
	1	2	3	4	5
.50 mean avg utility	.71	.68	.71	.67	.76
.50 mean utility	1.50	1.38	1.54	1.33	1.78
.60 mean avg utility	.76	.82	.77	.74	.72
.60 mean utility	1.37	1.63	1.43	1.26	1.18
.70 mean avg utility	.81	.81	.80	.82	.82
.70 mean utility	1.18	1.21	1.17	1.20	1.20
.80 mean avg utility	.88	.88	.88	.88	.88
.80 mean utility	1.14	1.21	1.23	1.21	1.21

Table 6.10 explanation.

problem

- The problem in each group of Experiment 6.4 tests

mean new avg utility

- The mean of the 5 unsolvable CarSP average utilities generated from *column* problem.

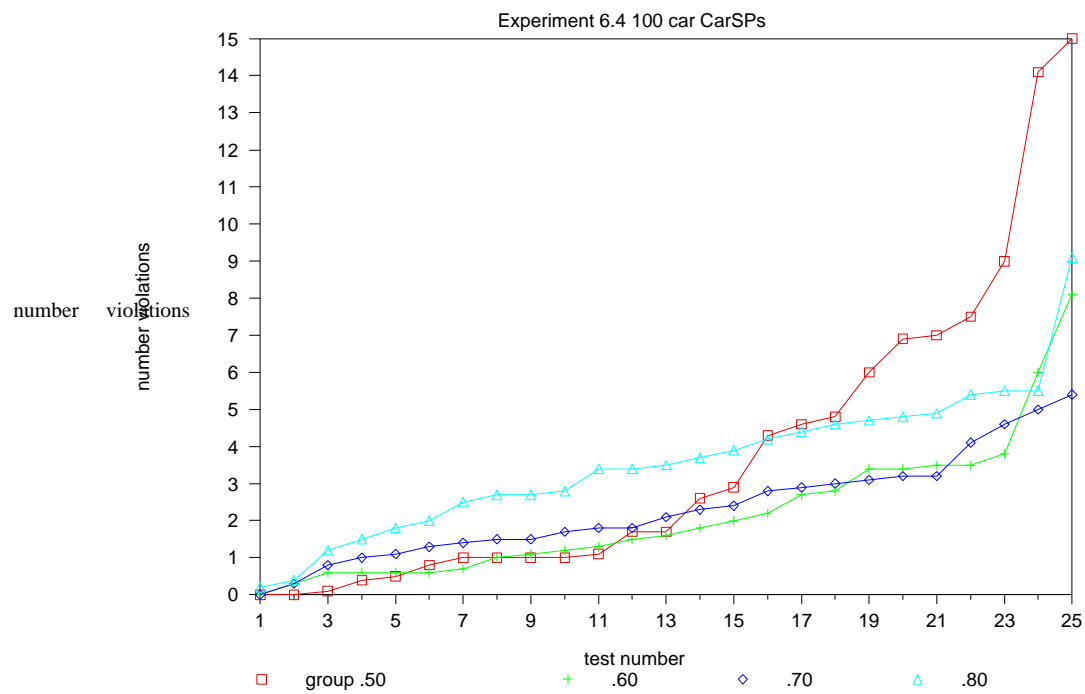
mean over-utilised utility

- The mean of the 5 unsolvable CarSPs u_m ($m = 1, 2, \dots, 5$) over-utilised utilities for each *column* problem.

Table 6.10 summarises the mean average utility for each group of generated CarSPs and mean over-utilised utility after the CarSPs are made unsolvable.

Figure 6.7 summarises the average minimal violations from the theoretical LB achieved for each test in the .50, .60, .70 and .80 groups. It shows that GAcSP performance can achieve near optimal results in terms of the theoretical LB.

Figure 6.7: Number of average violations for Experiment 6.4 CarSPs



6.4.2 Experiment 6.4 Discussion

Tables 6.9 and 6.10 summarise the results from testing GAcSP on unsolvable CarSPs, each with a single over-utilised option. The same GAcSP performance statistics were summarised from Experiment 6.4 as for Experiment 6.3. The theoretical LB fitness values were calculated using Equation 3.19 and the *maximum lower bound* by evaluating over-utilising Experiment 6.2 solution strings.

- ° **Definition 6.1 (Utility Ratio Tightness)** The utility ratio tightness of a CarSP can be measured as the number of spaces allowed in a CarSP schedule by the utility ratio. The tightness increases as the number of non-option spaces decreases, and is calculated as:

$$\text{utility ratio tightness} = N - \left(\frac{p_m}{q_m} \cdot N \right). \quad (6.1)$$

where there are N cars and p_m/q_m is the utility ratio for option m .

Claim 6.7 GAcSP can provide a near optimal performance in achieving the theoretical LB on very tight utility ratio unsolvable CarSPs.

Support 6.7 By over-utilising 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. Yet in general, the minimum and average violation solutions are close to the theoretical LB or within the theoretical and maximum lower bound range.

In Section 3.2.4 we developed a method for calculating a theoretical LB based upon an ideal schedule for a single over-utilised option. In addition, we calculated a maximum lower bound by using a solution found for each Experiment 6.2 CarSP, adding the extra options to the car types as described above and re-evaluating to discover the violation cost. These two values are represented in Figures 6.8, 6.9, 6.10, and 6.11 as the theoretical LB (min LB) and maximum lower bound (max LB). We can assume that the optimal minimal violation solutions for each group of tests must be within the range of these two values. Where the minimal solution achieves the theoretical LB (as is the case for most 1/2 ratio problems), we can be sure with some degree of confidence that the optimal minimal violation solution has been found. Equally, where a minimal violation solution lies outside the range as in the result for group .50 problem 1 in Figure 6.7, we can be sure that at least the maximum lower bound is achievable. However, even in this case the minimal violation solution could still be near optimal (i.e. so long as it is near to the maximum lower bound). Further, we cannot be confident that minimal violation solutions which are inside the range are optimal.

We can see from Table 6.9 results that an average 25% of the theoretical LB solutions are found with a further average of 64% within 3 violations. Several minimal violation results in Figure 6.7 are greater than the theoretical LB and closer to the maximum lower bound. We suggest that in these cases the theoretical LB is not achievable. It is therefore possible that

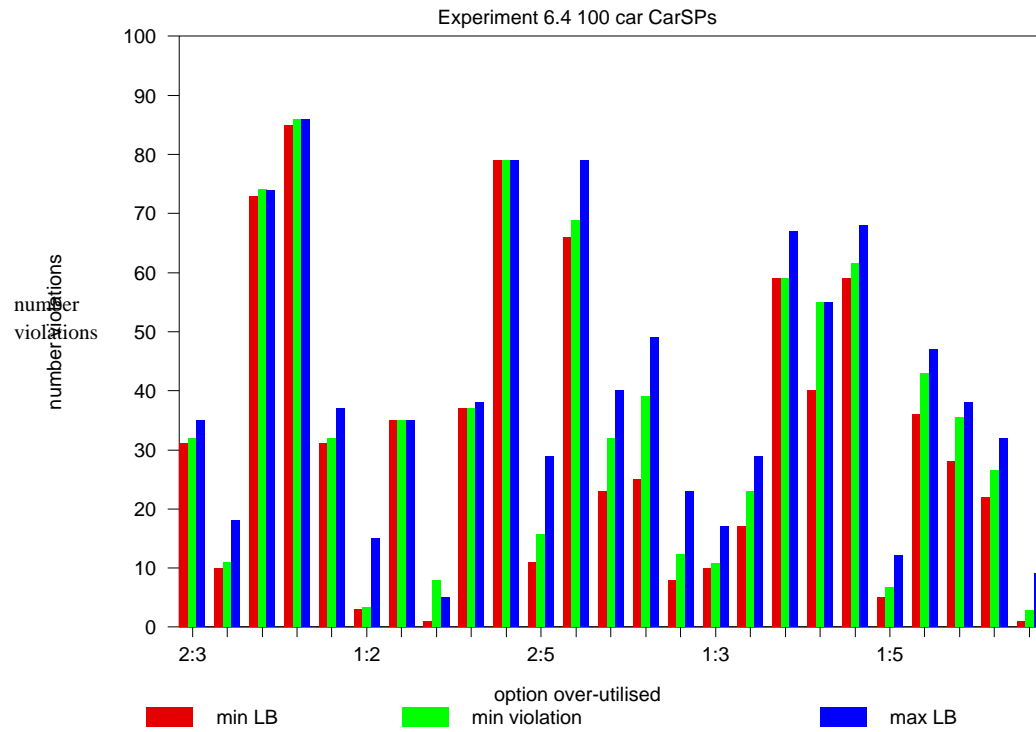
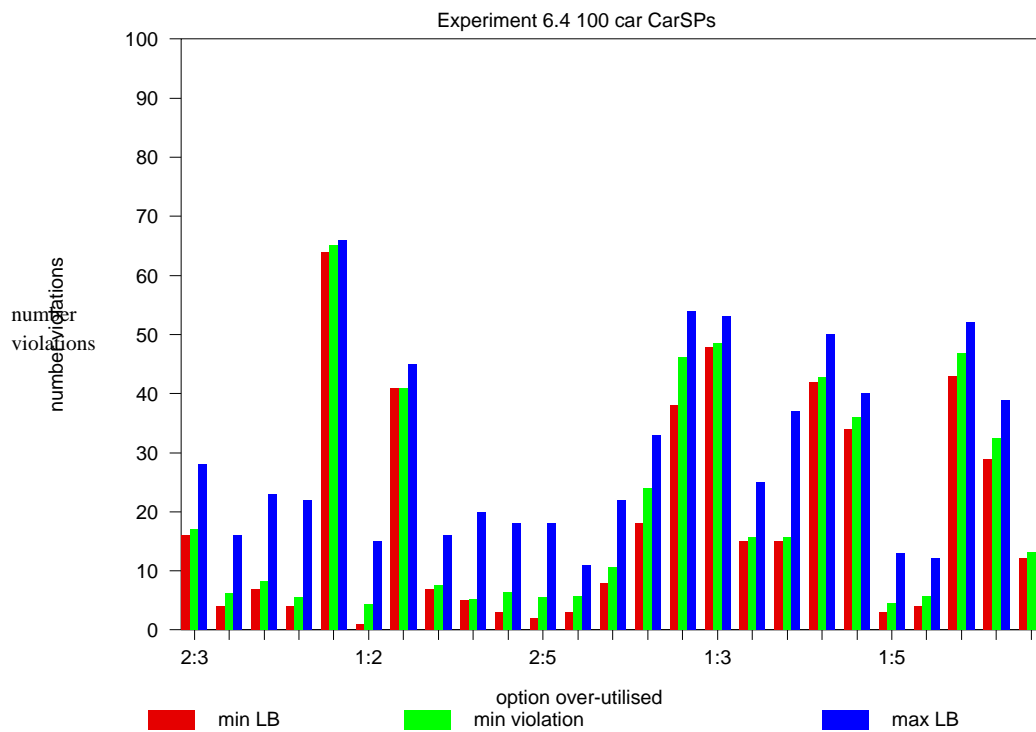
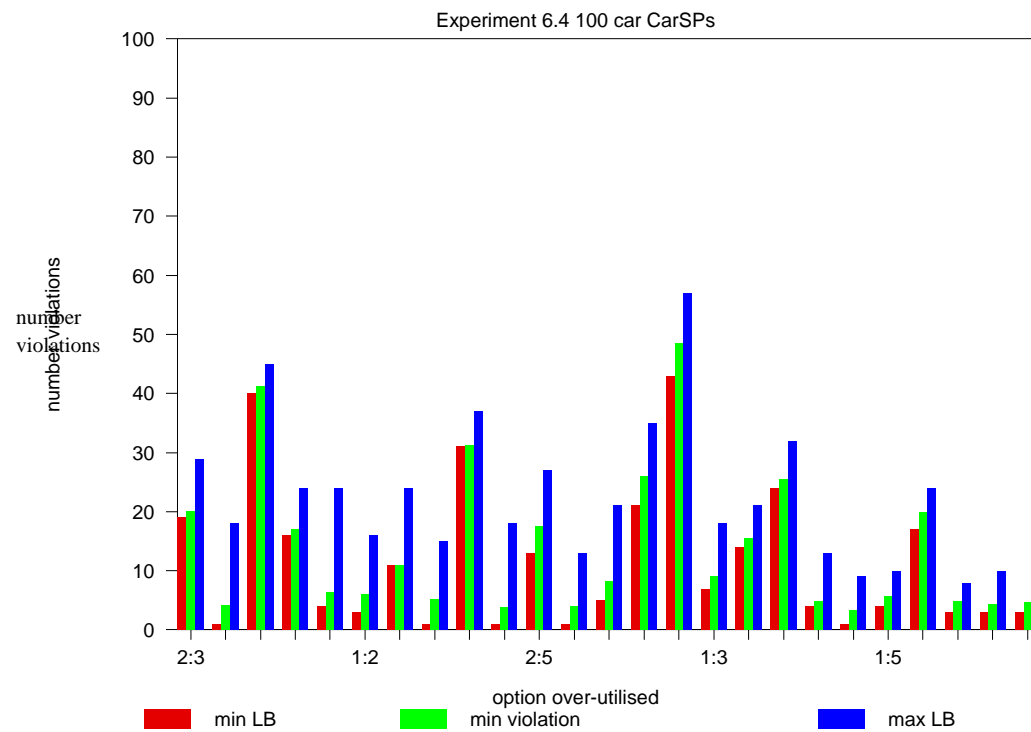
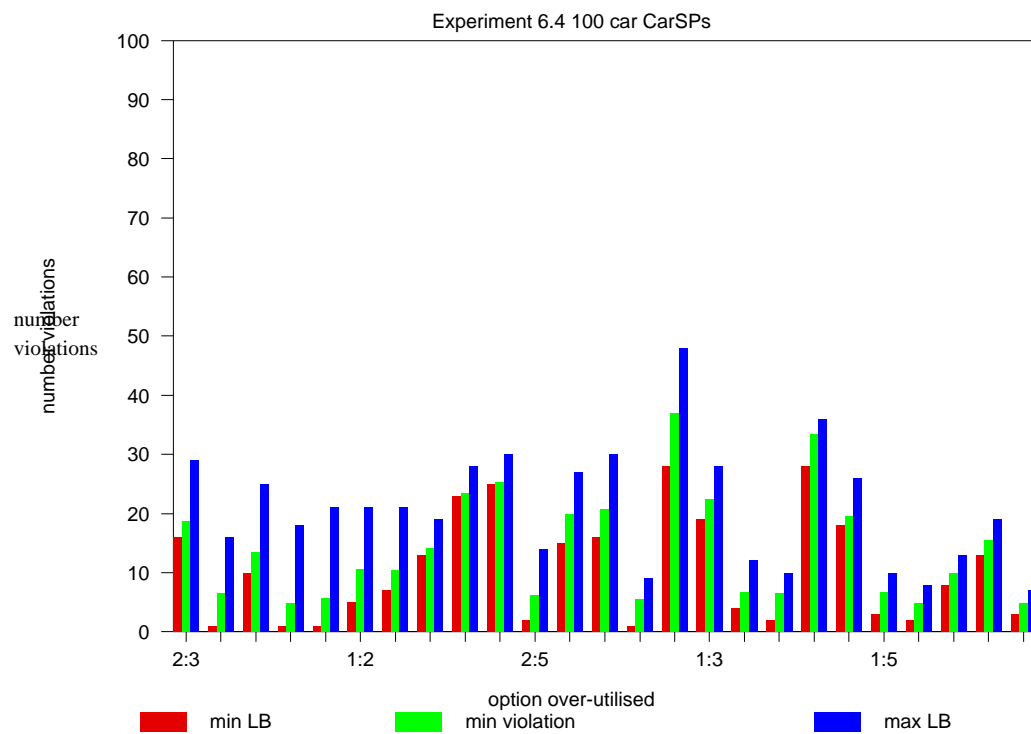
Figure 6.8: Summary average violations for group .50 CarSPs**Figure 6.9: Summary average violations for group .60 CarSPs**

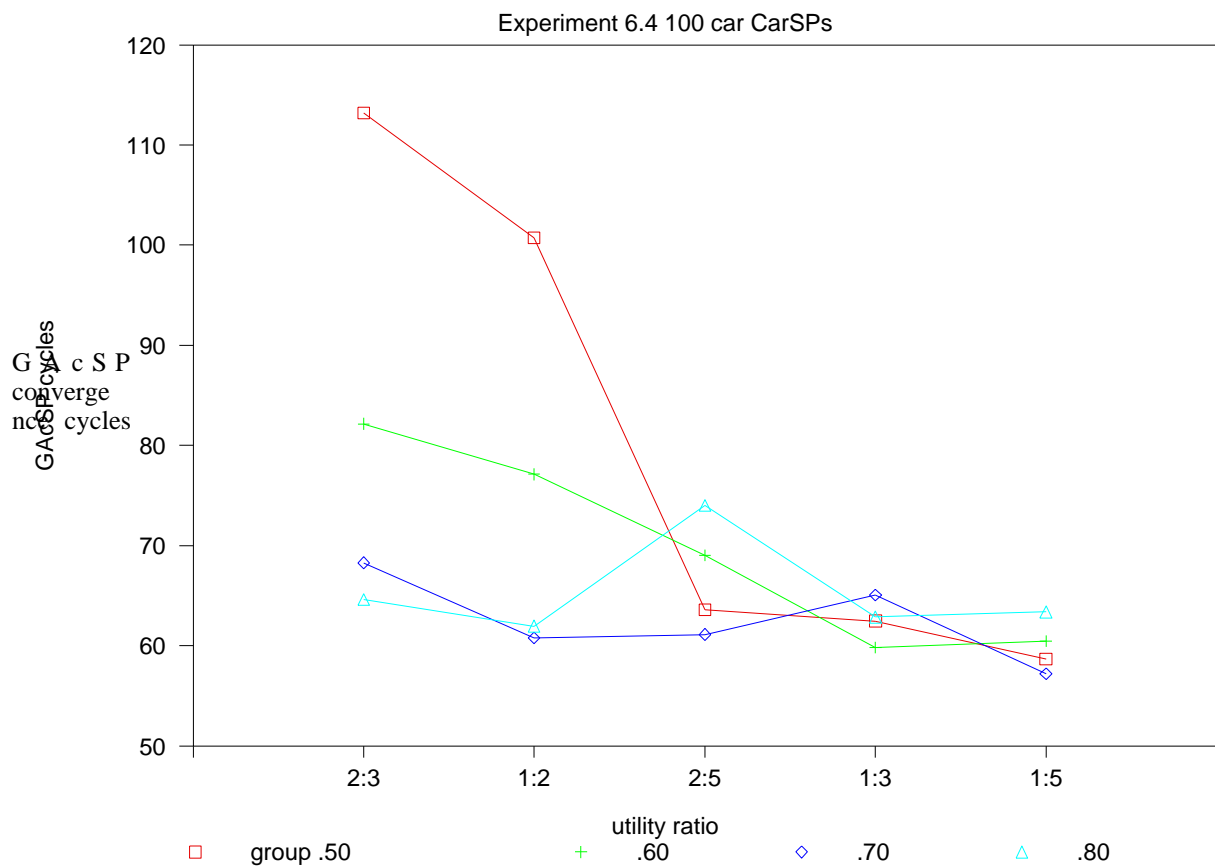
Figure 6.10: Summary average violations for group .70 CarSPs**Figure 6.11: Summary average violations for group .80 CarSPs**

the minimal violation solution found in these problems is the optimal or is a near optimal solution. ■

° **Claim 6.8** GAcSP can sustain the search in tackling unsolvable CarSPs by the crossover template mechanism exploiting fixed values at string positions due to tight utility ratio constraints.

Support 6.8 In Figure 6.12 we present the mean number of cycles to convergence for each utility ratio run (except a few runs which were terminated at the maximum 400 cycles). The

Figure 6.12: Summary Experiment 6.4 GAcSP cycle results



x -axis in Figure 6.12 represents decreasing utility ratio tightness, measured according to Definition 6.1 and Equation 6.1. In general we can see from Figure 6.12, the number of cycles for each test utility decrease as the utility ratio tightness decreases, demonstrating a positive correlation

between utility ratio tightness and GAcSP cycles. The curves for average utility tests .50, .60, and .70 demonstrate this correlation, but not so strongly with .80. Since we derived the .80 unsolvable CarSPs from Experiment 6.2 solvable .80 CarSPs and Experiment 6.2 results showed GAcSP found these solvable problems difficult to solve. The cycles to convergence for .80 unsolvable CarSPs is determined more by the average utility than utility ratio tightness. 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. GAcSP was able to exploit the *hard* position dependent constraints in sustaining the search with increasing utility ratio tightness.

In order to sustain the search, the crossover mechanism must use knowledge of constraints in a purposeful way. First consider the most successful option 1 utility ratio 1/2 tests, in achieving the theoretical LB (see Figures 6.8 to 6.11). Consider an example 100 CarSP where we have over constrained option 1 in a CarSP by 16 options - instead of the 50 cars requiring option 1 permitted by the 1/2 utility ratio we are required to sequence 66. To place 50 cars requiring option 1 in a schedule of 100 cars (ignoring other option capacity constraints) we need to separate each option 1 car with a car not requiring option 1. In fact, when sequencing 50 option 1 cars to satisfy the 1/2 utility ratio the only sequencing decision to be taken is whether to have the first car requiring option 1 in schedule position 100 or position 99 as in,

schedule position	100	99	98	97	...
requires option	Y	N	Y	N	...

or

schedule position	100	99	98	97	...
requires option	N	Y	N	Y	...

where "Y" indicates a car type requiring option 1 and "N" a car not requiring option 1. All cars which are sequenced following the placement of the first car at either position 100 or 99 must allow at least one non-option 1 car before placing the next option 1 car. In order to sequence 50 option 1 cars and satisfy the capacity constraint we can only use a single non-option 1 car to separate the option 1 cars. After these first 50 cars in the schedule

requiring option 1 have been sequenced, the remaining 16 cars need to be placed in such a way as to minimise the capacity violation for option 1. With the exception of placing an option 1 car at the end of the schedule (i.e. the special case mentioned in Section 3.2.4), there is no decision on placing the 15 option 1 cars of the total 66 left which will effect the degree of capacity violation. This principle applies also to Experiment 6.4 CarSPs ratio 1/3 where due to tight capacity constraints there are few alternative ways to position the maximum number of options allowed in a schedule.

In tightly constrained option ratio 1/2 and 1/3 CarSPs, the positioning of cars exceeding the maximum allowed by the option utility ratio has little effect on capacity violation. Yet, the absolute position of the over-utilised options in a schedule in respect of other options are important in achieving the theoretical LB. The crossover operator can use the position dependency due to tight constraints to form good building blocks. The crossover operator can allow GAcSP to creep towards the *optimal* theoretical LB by ensuring tightly positioned options are recorded on the binary templates.

Our second example test CarSP with over-utilised option 4 (utility ratio 2/5) has 43 cars requiring option 4 instead of 40 allowed by the ratio. The number of ways to position option 4 cars with utility ratio 2/5 is much greater. For example,

schedule position	10	9	8	7	6	5	4	3	2	1
example 1:	Y	Y	N	N	N	Y	Y	N	N	N
example 2:	Y	N	N	N	Y	Y	N	N	N	Y
example 3:	Y	N	Y	N	N	Y	N	Y	N	N
example 4:	Y	N	N	Y	N	Y	N	N	Y	N

represent alternative schedules for option 4 car types which satisfy the capacity constraint. However, our over-utilised option 4 example CarSP requires 3 more options than the capacity ratio allows, so in our example we add an extra 3 options as in

schedule	position	10	9	8	7	6	5	4	3	2	1
example 5:		Y	Y	N	N	N	Y	Y	Y	Y	Y
option 4	violation 3								V	V	V
example 6:		Y	N	N	Y	N	Y	Y	Y	Y	Y
option 4	violation 4							V	V	V	V
example 7:		N	Y	Y	Y	Y	Y	Y	N	N	Y
option 4	violation 5				V	V	V	V			V

We can see that only example 5 gives the theoretical LB violation of 3. Example 5 is derived from example 1 by adding an extra 3 options. Adding three extra options to examples 2 and 3 will not result in the theoretical LB being found. Therefore the difficulty faced by GAcSP in achieving the theoretical LB in test 2/5 requires selecting a combination of patterns which, when extra cars with options over the maximum are added, will result in a minimum violation. When tackling CarSPs with option 4 over-utilised there may only be one pattern of option to non-option cars which will produce a lower bound violation on the addition of extra options. Therefore options which have decreased tightness may allow more alternative arrangements in placing options in a schedule to satisfy the capacity constraint. The alternatives require more work from GAcSP to achieve the theoretical LB, but near optimal results are possible. ■

° **Claim 6.9** The GA component of GAcSP ensures robustness whilst the HC component adds a specialist ability.

Support 6.9 We can use an example unsolvable CarSP, where option 5 is over-utilised by a single extra option (20 options allowed according to ratio 1/5, yet 21 to sequence). In order to achieve the theoretical LB of 1 in this test problem, GAcSP is required to place this extra option at the end of the schedule where it will only violate one option. Also, the 20 option 5 allowed by the utility ratio must be correctly sequenced ensuring no capacity violation. The HC component can fine tune near optimal solutions to minimise the capacity violation. However, to sequence all option 5 in a schedule to satisfy the capacity constraint 1/5 is more difficult and beyond the localised ability of the HC. Since re-positioning a single option 5 will cause a violation, only re-positioning all option 5 can prevent a violation occurring. The following example partial schedule illustrates this difficulty

schedule	position	15	12	11	14	13	10	9	8	7	6	5	4	3	2	1
		Y	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N

There is extra space between option 5 at position 2 and 10. Instead of 4 no-option 5 spaces between position 2 and 10 there are 9. Moving a single option 5 into this space using HC will only create a capacity violation either for the option placed or at position 2. This obstacle can be overcome in two ways by re-sequencing all schedule options 5 from position 2 simultaneously or a planned sequence of individual changes. The UAX crossover in GAcSP can simultaneously re-position several options through the recombination of parent strings, and this can reduce the capacity violation but has no specific planning ability. ■

- **Conjecture 6.2** We would conjecture that in these unsolvable CarSPs with utility ratios $2/3$, $1/5$ and $2/5$ over-utilised, search methods which rely principally upon local information will easily get trapped in local minima.

Run-times shown in Figure 6.12 are longer for Experiment 6.4 tests of the same size and average utility as Experiment 6.2 from which they were derived, due to the runs terminating only after complete convergence. The intention was to ensure that the theoretical LB could not be improved upon, and to demonstrate typical run-times for example 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]. In which case, a compromise can be reached where optimality can be sacrificed for speed in unsolvable CarSPs.

6.5 Summary

We have tested GAcSP (using the hill-climber (HC)) on the car sequencing problem (CarSP). The problem of CarSP is to sequence cars requiring options into a schedule so that an assembly line of workstations teams can fit them. We test GAcSP's ability to cope with loose and tight constraints, problems of increasing size, and the performance of GAcSP on unsolvable problems. Our results are compared with fellow researchers and published results. The objective function for the algorithms calculated the number of options violated in a schedule. The GAcSP constant values of all the tests were a population size of 80 strings, 10% elite members, 4 offspring,

a maximum of 400 generations and a maximum of 30 CPU seconds for HC. For each test the number of solutions found, or minimum violation solution, and the time in CPU seconds to terminate were recorded and summarised.

The first tests (Experiment 6.2) were undertaken on solvable 100 car CarSPs with average utilities from .45 to .90. For each average utility, 10 solvable CarSPs were generated and 10 runs carried out on each problem. All CarSPs were generated by a program which provided a solution to each problem, where all capacity constraints were satisfied and the number of car types varied between $1 \leq k \leq 2^n$. GAcSP results were compared with a *heuristic repair* (HR) and HR combined with a single state *TABU* search (TABU).

- GAcSP can find more solutions to increasing .45 to .90 average utility CarSPs than the HR algorithm. Where we calculated a statistically significant difference between the number of solutions found by GAcSP and HR, demonstrating that GAcSP finds more solutions than HR for all average utility tightness CarSPs.
- The GAcSP strategy has achieved better average performance in finding solutions to average utility .45 to .90 tests than HR and TABU. If we calculate the average number of runs finding solutions for each algorithm, we find GAcSP has an average 77.2, HR 73.8 and TABU 71.8.
- The GAcSP performance (*robust*) in finding solutions to complex tightly constrained average utility CarSPs is better than HR and TABU because GAcSP does not depend entirely upon local search information. We discuss the fact that HR and TABU are more dependent for exploration upon local search space information at a single point than GAcSP, which uses a population of points. With increasing option interactions local information is less complete in guiding the search towards solutions and the starting points used are more important.
- GAcSP can provide more consistent near optimal performance on average minimal violation results for increasing average utility tightness in CarSPs than HR and TABU. The rate of increase in GAcSP average violation results for CarSPs is not as great as for HR and TABU. The mean of average minimal violation results of GAcSP, HR and TABU on

CarSPs are GAcSP 0.592, HR 1.097 and TABU 1.926.

- GAcSP has the opportunity to improve the quality of its performance on increasing average utility CarSPs which is not reduced to the extent of that shown by HR and TABU. We have the opportunity with GAcSP, by increasing the maximum improvement time for HC to improve the performance of GAcSP. However, if we allow more time to HR and TABU the performance is unlikely to be improved. Because HR and TABU work from a single search space point they are more likely to settle in local minima.

The second tests (Experiment 6.3) were undertaken on solvable CarSPs with 100, 120, 140, 160, 180 and 200 cars to sequence, and average utility ratios .50, .60, .70, and .80. There were 5 randomly generated CarSPs for each average utility ratio and 5 runs carried out on each problem.

- The performance of GAcSP in finding solutions to CarSPs is not significantly reduced by problem size. Although there is a slight reduction in the number of solutions with the increase in the number of cars, generally the performance of GAcSP is consistent. The loss in performance is not significant yet the increase in search space size for these CarSPs is significant.

We can make a limited comparison of our results with those reported by Parrello *et al.* [1986] who used an Automated Reasoning Program they called ITP to sequence 5 cars with 5 options. This took 35 minutes, and 15 minutes with ITP written using OPS5. Dincbas *et al.* [1988] have developed a Constraint Logic Programming Language (CHIP) which tackled solvable CarSPs. They reported that CHIP could sequence 100 car schedules with an average utilisation of .80 in under 60 seconds and 200 cars between 336 and 345 seconds. Although their approach showed good results it was restricted to solvable CarSPs only.

Our third set (Experiment 6.4) of tests were undertaken on unsolvable CarSPs, each with a single over-utilised option. We carried out tests on 4 groups of problems. Each group named .50, .60, .70, and .80 has 25 unsolvable CarSPs derived from our second test CarSPs. There were 10 runs for each unsolvable CarSP and therefore a total of 50 runs for each option and

average utility. In addition to the same GAcSP performance statistics as tests one and two, theoretical LB fitness values were calculated using Equation 3.19 and the *maximum lower bound* by evaluating over-utilising test one solution strings.

- GAcSP can provide a near optimal performance in achieving the theoretical LB on very tight utility ratio unsolvable CarSPs. On average 25% of the theoretical LB solutions are found, with a further average of 64% within 3 violations. Several minimal violation results are greater than the theoretical LB and closer to the maximum lower bound. We suggest that in these cases the theoretical LB is not achievable.
- GAcSP can sustain the search in tackling unsolvable CarSPs by the crossover template mechanism exploiting fixed values at string positions due to tight utility ratio constraints. 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. GAcSP was able to exploit the *hard* position dependent constraints in sustaining the search with increasing utility ratio tightness.
- The GA component of GAcSP ensures robustness whilst the HC component adds a specialist ability. The UAX crossover in GAcSP can simultaneously re-position several options through the recombination of parent strings, and this can reduce the capacity violation but relies upon HC for local improvement.

Chapter 7 Conclusion And Future Development

7.1 Contribution Of This Research

We have developed through this research a flexible heuristic strategy, which we call GAcSP. The GAcSP is a genetic algorithm and local improvement strategy which exploits PCSP features. The "engine" of GAcSP is a crossover operator (UAX) which remembers valuable crossover points and does not disrupt building blocks distributed over the parent strings. The crossover operator can exploit PCSP constraints in an efficient way. The power of the search is improved by the use of an underlying binary search space through an extended binary string representation. GAcSP has been tested on two instances of PCSPs the processors configuration problem (PCP), and the car sequencing problem (CarSP). These tests provide support to our claims that GAcSP is a generic PCSP solver, which can achieve optimal or near optimal solutions to solvable and unsolvable class of PCSPs. By tackling two problems, different to each other in their characteristics and requirements we learned about different GAcSP aspects. We can summarise the following discoveries from these tests:

(1) The PCP results demonstrate that the GA component of GAcSP can provide near optimal results to PCPs. When the GA component in GAcSP is combined with a hill-climber it can out-perform a special written program (i.e. AMP [Chalmers and Gregory, 1992]) for PCPs. The combination of GA with a hill-climber is a synergistic one which has improved the quality of solutions but at an extra computational cost. We also show that GAcSP has the potential for solving larger problems.

(2) GAcSP out-performed both HR and TABU on average, in finding solutions and minimising constraint violations to CarSPs and was not limited by being completely dependent upon local search information. The CarSP results show that GAcSP is not restricted to tackling solvable problems only, and that GAcSP can be effective in both loosely and tightly constrained problems. It is a robust search technique and is not prevented by problem size (i.e. 100 to 200 car CarSPs) from finding solutions in a reasonable time period. Through the action of the crossover operator, GAcSP can exploit problem 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 allowing scalability of problems. As larger problems are tackled the GA component can undertake more responsibility for the search. With larger search spaces the hill-climber depends more on the GA component providing points located within areas containing solutions. Unlike other optimisation techniques for PCSPs, GAcSP is a robust exploration strategy which does not easily get trapped in local minima. Therefore, GAcSP could provide a useful practical tool for tackling a class of combinatorial problems where current solution techniques are limited or infeasible. GAcSP is a general strategy which has achieved good results on both representative PCSPs with only problem specific data and domain specific evaluation functions. This research supports our research objective of developing a generic GA PCSP solver.

7.2 Further Developments

7.2.1 Using Diversity Information

In this section we outline a method for analysing the diversity of string values in GA populations. If we calculate the initial population diversity and progressive GAcSP populations we could use this information to help guide the search in two possible ways. The first is in the automatic control of GAcSP parameters. The main parameters for GAcSP are population size, number of offspring generated, number of elite members copied into the matepool and the HC time limit when switched on. These parameters control the rate of convergence for GAcSP and therefore the quality of solutions and time taken to achieve them. (We standardised the parameter values for experiments reported in this thesis based upon earlier work [Tsang and Warwick, 1989].) Secondly, better informed GAcSP decisions could be made at a number of points in the GA cycle. For example in the repair mechanism we could opt to repair the values shown to be missing in the population, in order to restore them. Other decision points are in the crossover mechanism, and HC. The use of population diversity to assist GA search has been used by Fang *et al.* [1993] in tackling Job-shop and Open-shop scheduling problems. Where they implemented a technique they called *gene-variance based operator targetting* which sampled statistical variance after every ten generations, and used the variance to choose the point of crossover

or mutation.

The underlying effect of population size, number of offspring, elite copies and HC time limit upon GAcSP is to change the diversity of the population. Both solution quality and the time taken to converge depend upon the rate of population diversity loss. We can quantify this loss by measuring the string values contained in the population at specific GAcSP intervals. We may need to consider the rise and fall of population values over successive generations due to the continuity of building blocks, when recording at specific time intervals. We referred earlier in the thesis to the idea of real-coded alphabets providing *finer grained* tools. The real-coded string representation used by GAcSP enables us to identify when string values at specific string positions are decreasing or increasing at a point in GAcSP processing.

The Walsh function analysis developed by Bethke [1972] and extended by Holland for measuring epistasis is compute intensive to implement and limited to fixed length binary strings. Davidor's technique depends upon a linear assumption where any fitness function can be reduced to a set of linearly independent partial fitness functions [Goldberg, 1988]. The method we use is to record in an array the number of string values at each string position in the population at a specific time. From this array we can calculate the population diversity using the standard deviation for each value at each string position. To do this we first define an $|v_i| \times N$ matrix A for $i = 1, 2, \dots, N$. The value a_{ji} represents the total number of PCSP value v_{ij} from the domain of PCSP variable i in population P_i i.e. $a_{ji} = |P_i|$.

We would expect a randomly generated initial population of a reasonable size (i.e. ≈ 50 to 100) to have at least one value in each column and row. That is, all variable values should be expressed at least once in the population of strings. But as the search progresses some values will be lost while others increase in number, until full convergence when only one row in each column will have a value. If the search has been successful we would expect the values in each column to represent a solution string or minimal violation string. We define a column average $cavg$ based upon the total number of values which can be expressed in a column divided by the domain size of the column. Further, the total number of values which can be expressed for any column i will be equal to the population size n . Thereby, we can

calculate the column i average $cavg_i$ as,

$$\text{column average } cavg_i = \frac{n}{|v_i|}. \quad (7.1)$$

For example, if we have a population of 30 strings where the domain size $|v_1| = 5$ then $cavg_1 = 30/5 = 6$. If all domains in a PCSP are of the same size,

$$|v_1| = |v_2| = \dots = |v_N|$$

then,

$$\{cavg_1 = cavg_2, \dots = cavg_N\}.$$

We can summarise these ideas in Table 7.1 which represents column 1 of a_1^t for a 30 string population at 3 GAcSP time periods; $t = 0$ (initial); $t = 10$; and $t = 30$ (full convergence). (We only consider a single string position to simplify the example):

(PCSP variable x_1)		
	x_1	x_1
v_{11}	6	10
v_{12}	6	3
(PCSP value) v_{13}	6	4
v_{14}	6	13
v_{15}	6	0
time t	0	10
total values n	30	30
column average $cavg_1$	6	6

Table 7.1: Population diversity string values (position $i = 1$)

The column average $cavg$ is based upon an *ideal* representation of values where all domain values are equally represented in a population of strings. Using the standard deviation we can quantify the deviation of values from this average. Summing the standard deviation for each

column in a^t will give us a single measure of population diversity Pd_t at specific time or cycle t , which can be calculated as follows:

$$\text{population diversity } Pd_t = \sum_{i=1}^N \sum_{j=1}^k (a_{ij}^t - \text{cavg}_i)^2. \quad (7.2)$$

Using Equation 7.2 we can calculate the population diversity for each a_1^t in Table 7.1: $Pd_0 = 0$; and $Pd_{10} = 78$. It should be possible to apply the calculation during a GAcSP run or after termination with recorded populations, depending on the frequency required. Figures 7.1, and 7.2 illustrates changing number of values at string positions for an initial 30 string population, and after 10 cycles for GAcSP tackling a 10 variable PCSP with domain size 5. If we look at Figure 7.1 we can see in the initial population that most values are equally represented, with no single value dominating. After 10 cycles Figure 7.2 shows that some values are beginning to dominate their string positions.

Figure 7.1: Population values count

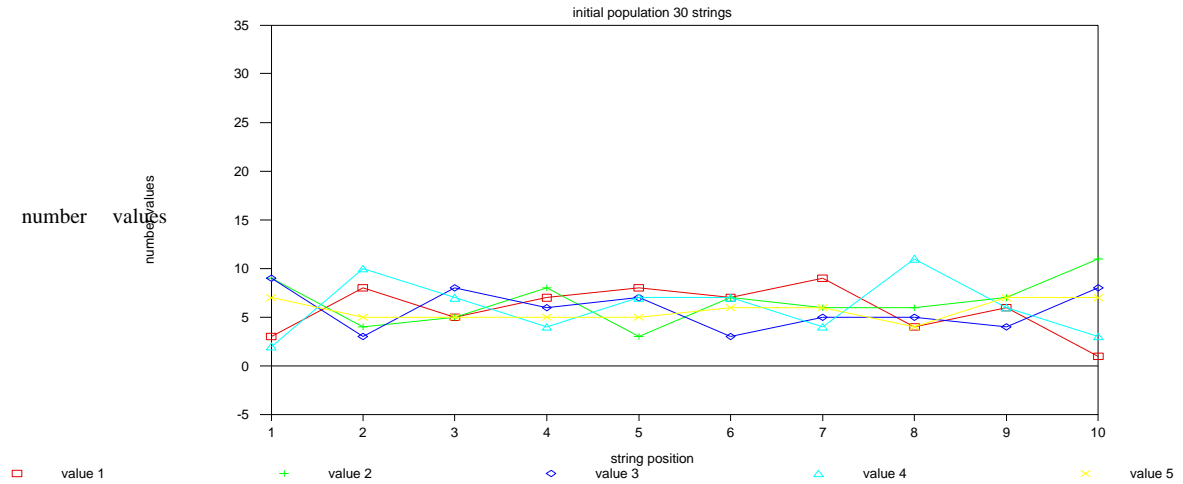
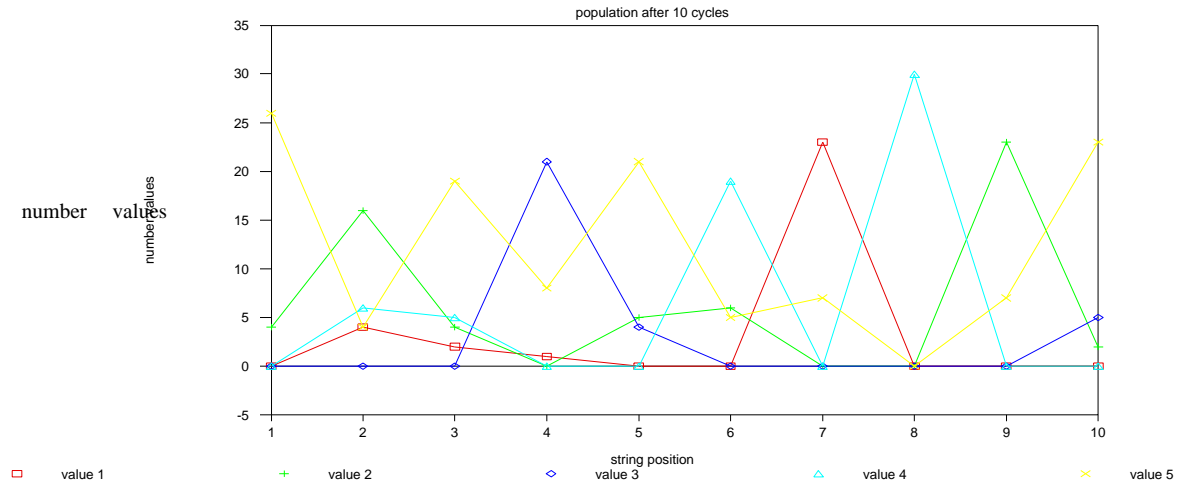


Figure 7.2: Population values count



The method outlined above for measuring population diversity is simple to implement and does not require intensive computing resources. The diversity Pd will quantify the effect of parameter changes as GAcSP progresses. For example, whether increasing the HC time limit will result in less or more diversity and what changes occur over specific GAcSP time periods.

7.2.2 Tackling Binary CSPs

The PCSPs tackled in this thesis, namely the PCP and CarSP are PCSPs which have constraints represented in the cost function. Both these problems required offspring strings to be repaired because of hard representation constraints. In the PCP representation these were dependent upon the valency of the processors and in the CarSP dependent on the production requirements. In earlier research [Tsang and Warwick, 1989] we were able to tackle binary CSPs which have a cost function defined by the CSP values, using a GA. Binary CSPs are an important class of problems not least because other n -ary constraint CSPs can be reduced to binary CSPs [Rossi, Petrie and Dhar, 1989]. If instead of a cost function based upon each CSP value having a cost, we define the GA fitness as a penalty function then GAcSP can tackle these also. A simple penalty function could be used which assigns a penalty violation cost of one to every binary constraint violated. This is the same technique of counting CarSP violation used in Experiments 6.2 and 6.3 tests which demonstrated GAcSP's success in finding solutions. With this approach the repair function is not needed, unless we can distinguish between binary constraints which must be satisfied (hard), while other constraints (soft) can be handled by the penalty function. This further development work will greatly extend the range of problems which can be tackled by GAcSP.

7.3 Summary and Conclusion

In this thesis GAcSP has been developed as a flexible heuristic strategy to tackle NP -hard partial constraint satisfaction problems (PCSPs). Domain knowledge can assist techniques used to tackle NP -hard problems but usually at a cost of becoming problem specific. GAcSP has been designed to exploit this knowledge for a class of problems (PCSPs) without losing a general problem solving performance. GAcSP is a combination of a modified standard GA and local improvement operator which has been designed to:

- Combine the advantages of a real-coded representation with those of a binary coded representation by utilising a binary template in the UAX crossover operator.
- Combine the GA with a local improvement technique (HC) which exploits features of PCSPs.
- Provide the advantages to tackling a class of problems without losing a general problem solving performance.
- Allow flexibility in approach to PCSPs with different hard and soft constraint requirements.
- Provide a reasonable compromise on the optimality of solutions and the time taken to find them.

Using GAcSP we demonstrated that GA can be adapted to become suitable for tackling constraint based problems and is not necessarily restricted to problems with numerical constraints or which only have a few constraints. Our analysis of constraint interaction or epistasis in PCSPs has shown why standard GAs find such problems difficult and how GAcSP can exploit tight constraints in finding solutions.

Program GA1 has been tested against two difficult problems, the processors configuration problem (PCP) and car sequencing problem (CarSP). Test problems can limit the scope of any claims made from results. However, we have tried to test two important characteristics of partial constraint satisfaction problems (PCSP), namely constraint interaction and the problem of combinatorial explosion. We tested these by systematically increasing the number of constraints and the number of variables in our example PCSPs. These empirical tests are needed to understand program behaviour, particularly performance under certain conditions. The performance of GA1 on both problems have been compared to alternative approaches to each problem. In general, an algorithm might be useful in one aspect of solving problems but poor in performance on others. In summarising the results for our tests we have focused on two main performance measures, the "best" (in terms of minimising or maximising the objective function) and average solution quality and the time taken to achieve it. Furthermore, because GAcSP makes no

distinction between solvable and unsolvable problems this can prevent time being used trying to find solutions to unsolvable problems which cannot be determined *a priori*.

There are a number of important issues which have been raised in this research, one of which is the paradox between the success of GAs using real-coded representations and Schemata Theory or building block hypothesis. On the one hand, this research to some extent contributes to this paradox by demonstrating successful search using a real-coded representation. On the other, we have tried to show by analysis that this success is partly due to the use of a binary template mechanism creating an underlying binary search space. We discussed the limitations with the Schemata Theorem and binary representations and considered alternative theories. Furthermore, it is important to consider the GA system, in particular the relationship between crossover and representation and what influence this has on the schemata that are formed. It is the correlation between schemata and string fitness which will determine the ultimate success of a GA on a specific problem.

Designing strategies for constraint satisfaction problems is an important step towards establishing AI as a practical approach to tackling real problems. This is particularly important for the many problems which fall in the domain of scheduling, where production cost reduction can be significant and the quality of service improved. The size and complexity of scheduling problems justifies combining different approaches in creating synergistic strategies. Finally, we hope this research has provided some insight into using an adaptive approach to tackling CSPs, and generated new ideas which can be exploited by further work.

Summary Of Important Symbols

(page numbers indicate first or defining occurrence of symbol)

#	metasymbol which stands for 0 or 1	9
C	set of constraints on an arbitrary subset of variables in Z , restricting the values that they can take together	3
D	set of discrete domains for each variable in Z $\{v_1, v_2, \dots, v_N\}$	3
g	function mapping PCSP solution tuples into numeric integers	3
H	schema	9
i	index for string elements and PCSP variable domain value	4
j	index for string elements	23
k	cardinality	4
l	length of string S or S_{bin}	6
n	number of strings in P	6
N	number of PCSP variables	3
ρ	probability	9
P	population of n strings S or S_{bin}	6
S	string of l non-binary elements	36
S_{bin}	string of l binary elements	6
t	period or unit of time	6
v	PCSP variable value indexed by i and j	4
Z	set of variables $\{x_1, x_2, \dots, x_N\}$	3

Appendix A HR and TABU Pseudo Code

Pseudo code for the heuristic repair (HR) and Tabu search algorithm (TABU). (Supplied by Kangmin Zhu.)

```

PROCEDURE TABU_CSP_1(Z, D, C, IterationLimit, TabuLengthLimit)
BEGIN
  FOR each variable xi in Z in a random order DO
    BEGIN
      xi the value which in Dx which involves in the least number
        of conflicts at the time, break ties randomly;
      Tabu[i] empty list;
    END;
  k 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 k ≥ IterationLimit;
END /* of TABU_CSP_1 */

```

When TabuLengthLimit = 0, TABU_CSP_1 becomes a simple HR algorithm;

and when TabuLengthLimit = 1, TABU_CSP_1 is a one-state Tabu algorithm.

Bibliography

Alberts B., Bray D., Lewis J., Raff M., Roberts K. and Watson J.D. (1983) *Molecular Biology of the Cell*. New York, USA: Garland Publishing Inc.

Antonisse J.H. (1989) *A New Interpretation of Schema Notation that Overturns the Binary Encoding Constraint*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 86-91.

Antonisse J.H. and Keller K.S. (1987) *Genetic Operators For High-Level Knowledge Representations*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 69-76.

Axelrod R. (1987) *The Evolution of Strategies in the Iterated Prisoner's Dilemma*. In *Genetic Algorithms and Simulated Annealing* (ed. L. Davis), London, UK: Pitman. pp. 32-41.

Bagchi S., Uckun S., Miyabe Y. and Kawamura K. (1991) *Exploring Problem-Specific Recombination Operators for Job Shop Scheduling*. Genetic algorithms and their applications: Proceedings of the Fourth International Conference, June 13-16 1991, (eds. R.K. Belew and L.B. Booker), University of California, San Diego, USA: Morgan Kaufmann. pp. 10-17.

Bagley J.D. (1967) *The behavior of adaptive systems which employ genetic and correlation algorithms*. (Doctoral dissertation, University of Michigan). Dissertation Abstracts International 28 (12), 1967, 5106B. (University Microfilms No. 68-7556).

Baker J.E. (1985) *Adaptive Selection Methods for Genetic algorithms*. Genetic algorithms and their applications: Proceedings of the First International Conference, July 24-26 1985, (ed. J.J. Grefenstette), Carnegie-Mellon University, Pittsburgh, USA: Lawrence Erlbaum Associates. pp. 101-111.

Bethke A.D. (1981) *Genetic Algorithms as Function Optimizers*. Doctoral dissertation, University of Michigan. Dissertation Abstracts International 41 (9), January 1981, 3503B. University Microfilms No. 8106101.

Bickel A.S. and Bickel R.W. (1987) *Tree Structured Rules in Genetic Algorithms*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 77-81.

Biggs N. (1974) *Algebraic Graph Theory, Cambridge Tracts in Math.* 1974, no 67. London, UK: Cambridge University Press.

Booker L.B. (1987) *Improving Search in Genetic algorithms. In Genetic algorithms and Simulated Annealing* (ed. L. Davis), London, UK: Pitman. pp. 61-73.

Booker L.B., Goldberg D.E. and Holland J.H. (1989) *Classifier systems and genetic algorithms*. Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, September 1989, Los Altos, California, USA: Morgan Kaufmann. pp. 235-282.

Bosworth J., Foo N. and Zeigler B.P. (1972) *Comparison of genetic algorithms with conjugate gradient methods*. Technical Report CR-2093, Washington, DC, USA: National Aeronautics and Space Administration.

Bounds D.G. (1987) *New optimization methods from physics and biology*. Nature 329, September 17 1987, pp. 215-219.

Bramlette M.F. and Bouchard E.E. (1991) *Genetic Algorithms in Parametric Design of Aircraft. In Handbook of Genetic Algorithms* (ed. L. Davis), New York, USA: Van Nostrand Reinhold. pp. 109-123.

Bridges C. and Goldberg D.E. (1987) *An Analysis of Reproduction and Crossover in a Binary-Coded Genetic Algorithm*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 9-13.

Brown D.E., Huntley C.L. and Spillane A.R. (1989) *A Parallel Genetic Heuristic for the Quadratic Assignment Problem*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 406-415.

Chalmers A. and Gregory S. (1992) *Constructing Minimum Path Configurations for Multiprocessor Systems*. Technical Report CSTR-92-12, April 1992, University of Bristol, Computer Science Department, UK.

Cleveland G.A. and Smith S.F. (1989) *Using Genetic Algorithms to Schedule Flow Shop Releases*. Genetic algorithms and their applications: Proceedings of the Third International Conference June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 160-169.

Davidor Y. (1991) *Genetic Algorithms And Robotics: A Heuristic Strategy for Optimization*. Singapore: World Scientific Publishing Co.

Davidor Y. (1991) *A Genetic Algorithm Applied To Robot Trajectory Generation*. In *Handbook of Genetic Algorithms* (ed. L. Davis), New York, USA: Van Nostrand Reinhold. pp. 144-165.

Davis L. (1985) *Applying adaptive algorithms to epistatic domains*. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, August 18-23 1985, Los Angeles, California, USA: Morgan Kaufmann. pp. 162-164.

Davis L. (1989) *Adapting Operator Probabilities in Genetic Algorithms*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 61-69.

Davis L. (1991) *A Genetic Algorithms Tutorial*. In *Handbook of Genetic Algorithms* (ed. L. Davis), New York, USA: Van Nostrand Reinhold. pp. 1-101.

Davis L. and Coombs S. (1987) *Genetic Algorithms and Communication Link Speed Design: Theoretical Considerations*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA:

Lawrence Erlbaum Associates. pp. 252-256.

Davis L. and Steenstrup M. (1987) *Genetic Algorithms and Simulated Annealing: An Overview*. In *Genetic Algorithms and Simulated Annealing* (ed. L. Davis), London, UK: Pitman. pp. 1-11.

Deb K. and Goldberg D.E. (1989) *An Investigation of Niche and Species Formation in Genetic Function Optimization*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 42-50.

De Jong K.A. (1975) *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), August 1975, 5140B. (University Microfilms No. 76-9381).

De Jong K.A. (1980) *Adaptive System Design: A Genetic Approach*. Proceedings of the IEEE Transactions Conference on Systems, Man, and Cybernetics, vol 10, no 9, September 1980, pp. 566-574.

De Jong K.A. (1985) *Genetic Algorithms: A 10 Year Perspective*. Genetic algorithms and their applications: Proceedings of the First International Conference, July 24-26 1985, (ed. J.J. Grefenstette), Carnegie-Mellon University, Pittsburgh, USA: Lawrence Erlbaum Associates. pp. 169-177.

De Jong K.A. (1987) *On Using Genetic Algorithms to Search Program Spaces*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 210-216.

De Jong K.A. (1988) *Learning with Genetic Algorithms: An Overview*. Machine Learning vol 3, Morgan Kaufmann, 121-138.

De Jong K.A. and Spears W.M. (1989) *Using Genetic Algorithms to Solve NP-Complete Problems*. Genetic algorithms and their applications: Proceedings of the Third International Conference,

June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 124-132.

Dincbas M., Simonis H. and van Hentenryck P. (1988) *Solving the Car Sequencing Problem in Constraint Logic Programming*. Proceedings of the European Conference on Artificial Intelligence, pp. 290-295.

Eshelman L.J., Caruana R.A. and Schaffer J.D. (1989) *Biases in the Crossover Landscape*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 10-19.

Fang H-L., Ross P. and Corne D. (1993) *A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Re-scheduling and Open-Shop Scheduling Problems*. Genetic algorithms and their applications: Proceedings of the Fifth International Conference, July 17-22 1993, (ed. S. Forrest), University of Illinois, USA: Morgan Kaufmann. pp. 375-382.

Fogel L.J., Owens A.J. and Walsh M.J. (1966) *Artificial intelligence through simulated evolution*. New York, USA: John Wiley and Sons.

Frantz D.R. (1972) *Non-Linearities in Genetic Adaptive Search*. (Doctoral dissertation, Department Computer and Communication Sciences, University of Michigan, Ann Arbor). Dissertation Abstracts International 33 (11), 1972, 5240B-5241B. (University Microfilms No. 73-11,116).

Freidman G.J. (1959) *Digital simulation of an evolutionary process*. General Systems Yearbook 4, pp. 171-184.

Glover D.E. (1987) *Solving A Complex Keyboard Configuration Problem Through Generalized Adaptive Search*. In *Genetic Algorithms and Simulated Annealing* (ed. L. Davis), London, UK: Pitman. pp. 12-31.

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

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

Goldberg D.E. (1985) *Dynamic System Control using Rule Learning and Genetic Algorithms*. Proceedings of the Ninth International Joint Conference on Artificial Intelligence vol 1, August 18-23 1985, Los Angeles, California, USA: Morgan Kaufmann. pp. 588-592.

Goldberg D.E. (1987) *Simple Genetic algorithms and the Minimal, Deceptive Problem*. In Genetic Algorithms and Simulated Annealing (ed. L. Davis), London, UK: Pitman. pp. 74-88.

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

Goldberg D.E. (1989) *Zen and the Art of Genetic Algorithms*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 80-85.

Goldberg D.E. (1989) *Sizing Populations for Serial and Parallel Genetic Algorithms*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 70-79.

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

Goldberg D.E. and Lingle R.Jnr. (1985) *Alleles, Loci, and the Traveling Salesman Problem*. Genetic algorithms and their applications: Proceedings of the First International Conference, July 24-26 1985, (ed. J.J. Grefenstette), Carnegie-Mellon University, Pittsburgh, USA: Lawrence Erlbaum Associates. pp. 154-159.

Goldberg D.E. and Richardson J.T. (1987) *Genetic Algorithms with Sharing for Multimodal Function Optimization*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 41-49.

Gorges-Schleuter M. (1989) *ASPARAGOS An Asynchronous Parallel Genetic Optimization Strategy*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 422-427.

Greene D.P. and Smith S.F. (1987) *A Genetic System for Learning Models of Consumer Choice*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 217-223.

Grefenstette J.J. (1987) *Incorporating Problem Specific Knowledge into Genetic algorithms*. In *Genetic algorithms and Simulated Annealing* (ed. L. Davis), London, UK: Pitman. pp. 42-60.

Grefenstette J.J. (1988) *Credit Assignment in Genetic Learning Systems*. Proceedings of the National Conference on Artificial Intelligence, 1988, pp. 596-610.

Grefenstette J.J. (1991) *Strategy Acquisition with Genetic Algorithms*. In *Handbook of Genetic Algorithms* (ed. L. Davis), New York, USA: Van Nostrand Reinhold. pp. 186-201.

Grefenstette J.J. and Baker J.E. (1989) *How Genetic Algorithms Work: A Critical Look at Implicit Parallelism*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 21-27.

Grefenstette J.J., Gopal R., Rosmaita B.J. and Van Gucht D. (1985) *Genetic Algorithms for the Traveling Salesman Problem*. Genetic algorithms and their applications: Proceedings of the First International Conference, July 24-26 1985, (ed. J.J. Grefenstette), Carnegie-Mellon University, Pittsburgh, USA: Lawrence Erlbaum Associates. pp. 160-168.

Grefenstette J.J. and Pettey C.B. (1986) *Approaches to Machine Learning with Genetic Algorithms*. Proceedings of the IEEE Transactions Conference on Systems, Man and Cybernetics vol 1, October 14-17 1986, New York, USA: IEEE. pp. 55-60.

Holland J.H. (1973) *Genetic algorithms and the optimal allocations of trials*. SIAM Journal of Computing vol 2, no 2, 88-105.

Holland J.H. (1975) *Adaptation in Natural and Artificial Systems*. The University of Michigan Press.

Holland J.H. (1986) *Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems*. Machine Learning vol 2, no 20, (eds. R.S. Michalski., J.G. Carbonnel and T.M. Mitchell), 1986, Los Altos, California, USA: Morgan Kaufmann. pp. 593-623.

Holland J.H. (1987) *Genetic Algorithms and Classifier Systems: Foundations and Future Directions*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 82-89.

Holland J.H., Holyoak K.J., Nisbett R.E. and Thagard P.R. (1987) *Classifier Systems, Q-Morphisms, and Induction*. In *Genetic Algorithms and Simulated Annealing* (ed. L. Davis), London, UK: Pitman. pp. 116-128.

Holland J.H., Holyoak K.J., Nisbett R.E. and Thagard P.R. (1987) *Induction: Processes of inference, learning, and discovery*. Cambridge, Massachusetts, USA: MIT Press.

Jog P., Suh J.Y. and Gucht D.V. (1989) *The Effects of Population Size, Heuristic Crossover and Local Improvement on a Genetic Algorithm for the Traveling Salesman Problem*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 110-115.

Kadaba N. and Nygard K.E. (1990) *Improving the Performance of Genetic Algorithms in Automated Discovery of Parameters*. Proceedings of the International Conference on Machine Learning, June 1990, Austin, USA.

Kadaba N., Nygard K.E. and Juell P.L. (1991) *Integration of Adaptive Machine Learning and Knowledge-Based Systems for Routing And Scheduling Applications*. Proceedings of the International

Expert Systems with Applications Journal vol 2. 15-27.

Kumar V. (1992) *Algorithms for Constraint Satisfaction Problems: A Survey*. AI magazine, vol 13, no 1, Spring 1992, pp. 32-44.

Lenat D.B. (1983) *The Role of Heuristics in Learning by Discovery: Three Case Studies*. Machine Learning vol 9: An Artificial Intelligence Approach (eds. R.S. Michalski., J.G. Carbonnel, and T.M. Mitchell), 1983, Palo Alto, California, USA: Tioga. pp. 286-305.

Lengauer T. (1990) *Combinatorial Algorithms for Integrated Circuit Layout*. Chichester, UK: John Wiley & Sons Ltd.

Liepins G.E., Hilliard M.R., Palmer M. and Morrow M. (1987) *Greedy Genetics*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 90-99.

Lin S. and Kernighan B.W. (1973) *An Effective Heuristic Algorithm for the Traveling Salesman Problem*. Operations Research 21, 1973, pp. 498-516.

Mackworth A.K. (1977) *Consistency in Networks of Relations*. Artificial Intelligence, vol 8, 99-118.

Mason A.J. (1991) *Partition Coefficients, Static Deception and Deceptive Problems for Non-Binary Alphabets*. Genetic algorithms and their applications: Proceedings of the Fourth International Conference, June 13-16 1991, (eds. R.K. Belew and L.B. Booker), University of California, San Diego, USA: Morgan Kaufmann. pp. 210-214.

Mauldin M.L. (1984) *Maintaining Diversity in Genetic Search*. Proceedings of the National Conference on Artificial Intelligence vol 1, 1984, pp. 247-250.

Meseguer P. (1989) *Constraint Satisfaction Problems: An Overview*. AI Communications, vol 2, no 1, March 1989, pp. 3-17.

Michalewicz Z. and Janikow C.Z. (1991) *Handling Constraints in Genetic algorithms*. Genetic algorithms and their applications: Proceedings of the Fourth International Conference, June 13-16 1991, (eds. R.K. Belew and L.B. Booker), University of California, San Diego, USA: Morgan Kaufmann. pp. 151-157.

Michalewicz Z., Vignaux G.A. and Groves L.J. (1989) *Genetic Algorithms for Optimization Problems*. Proceedings of the Eleventh New Zealand Computer Conference, August 16-18 1989, Wellington, New Zealand. pp. 211-223.

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

Mühlenbein H. (1989) *Parallel Genetic Algorithms, Population Genetics, and Combinatorial Optimization*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 416-421.

Nadel B. (1990) *Some Applications of the Constraint Satisfaction Problem*. Technical Report CSC-90-008, Department of Computer Science, Wayne State University, USA.

Oliver I.M., Smith D.J. and Holland J.R.C. (1987) *A Study of Permutation Crossover Operators on the Traveling Salesman Problem*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 224-230.

Parrello B.D. (1988) CAR WARS: AI Expert, January 1988, pp. 60-64.

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

Pettit E. and Swigger K.M. (1983) *An Analysis of Genetic-Based Pattern Tracking and Cognitive-Based Component Models of Adaptation*. Proceedings of the Eighth National Conference on Artificial Intelligence, August 1983, pp. 327-332.

Prior D., Norman M., Radcliffe N.J. and Clarke L. (1989) *What Price Regularity ?* Edinburgh Concurrent Supercomputer Project, January 25 1989.

Radcliffe N.J. (1989) *Genetic Neural Networks on MIMD Computers (compressed edition)*. (Doctoral dissertation), 1990. University of Edinburgh.

Radcliffe N.J. (1989) *Early Clustering Around Optima*. University of Edinburgh Reprint 89/457, February 8 1989.

Rendell L.A. (1983) *A Doubly Layered Genetic Penetrance Learning System*. Proceedings of the Eighth National Conference on Artificial Intelligence, August 1983, pp. 342-347.

Richardson J.T., Palmer M.R., Liepins G.E. and Hilliard M.R. (1989) *Some Guidelines for Genetic Algorithms with Penalty Functions*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 191-197.

Robertson G.G. (1987) *Parallel Implementation of Genetic Algorithms in a Classifier System*. In *Genetic Algorithms and Simulated Annealing* (ed. L. Davis), London, UK: Pitman. pp. 129-140.

Rosenberg R.S. (1967) *A simulation of genetic populations with biochemical properties*. (Doctoral dissertation, University of Michigan). Dissertation Abstracts International 28 (7), 1967, 2732B. (University Microfilms No. 67-17,836).

Schaffer D.J. (1984) *Some Experiments in Machine Learning using Vector Evaluated Genetic Algorithms*. (Unpublished doctoral dissertation), 1984 December. Nashville: Vanderbilt University, Department of Computer Science.

Schaffer D.J. and Grefenstette J.J. (1985) *Multi-Objective Learning via Genetic Algorithms*. Proceedings of the Ninth International Joint Conference on Artificial Intelligence vol 1, August 18-23 1985, Los Angeles, California, USA: Morgan Kaufmann. pp. 593-595.

Schaffer D.J. (1987) *Some Effects of Selection Procedures on Hyperplane Sampling by Genetic Algorithms*. In *Genetic Algorithms and Simulated Annealing* (ed. L. Davis), London, UK: Pitman. pp. 89-103.

Schaffer D.J. and Morishima A. (1987) *An Adaptive Crossover Distribution Mechanism for Genetic Algorithms*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 36-40.

Selfridge O.J. (1959) *Pandemonium: A paradigm for learning*. Proceedings of the Symposium on the Mechanization of Thought Processes, 1959, pp. 511-529.

Shaefer C.G. (1987) *The ARGOT Strategy: Adaptive Representation Genetic Optimizer Technique*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 50-58.

Siedlecki W. and Sklansky J. (1989) *Constrained Genetic Optimization via Dynamic Reward-Penalty Balancing and Its use in Pattern Recognition*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 141-150.

Skorin-Kapov J. (1990) *Tabu Search Applied to the Quadratic Assignment Problem*. Operations Research Society of America vol 2, no 1, 1990. (ORSA) Journal on Computing, 33-45.

Smith S.F. (1980) *A learning system based on genetic adaptive algorithms*. (Unpublished doctoral dissertation), December 1980. University of Pittsburgh.

Smith S.F. (1983) *Flexible Learning of Problem Solving Heuristics through Adaptive Search*. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, August 1983,

Karlsruhe, West Germany: Morgan Kaufmann. pp. 422-425.

Suh J.Y. and van Gucht D. (1987) *Incorporating Heuristic Information into Genetic Search*. Genetic algorithms and their applications: Proceedings of the Second International Conference, July 28-31 1987, (ed. J.J. Grefenstette), Cambridge, Massachusetts, USA: Lawrence Erlbaum Associates. pp. 100-107.

Syswerda G. (1989) *Uniform Crossover in Genetic Algorithms*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 2-9.

Syswerda G. (1991) *Schedule Optimization Using Genetic Algorithms*. In *Handbook of Genetic Algorithms* (ed. L. Davis), New York, USA: Van Nostrand Reinhold. pp. 332-349.

Tsang E.P.K. and Warwick T. (1989) *Applying Genetic algorithms to Constraint Satisfaction Optimisation Problems*. Proceedings of the European Conference on Artificial Intelligence. Stockholm, Sweden.

Tsang E.P.K. and Warwick T. (1989) *Applying Genetic Algorithms to Constraint Satisfaction Optimization Problems*. Technical Report CSM-139, December 1989, University of Essex, Department of Computer Science, UK.

Tsang E.P.K. (1993) *Foundations of Constraint Satisfaction*. London, UK: Academic Press Limited, Harcourt Brace & Company, Publishers.

Vose M.D. (1991) *Generalizing the notion of schema in genetic algorithms*. Artificial Intelligence 50 (3), Research Note. August 1991, pp. 385-396. North-Holland.

Vose M.D. and Liepins G.E. (1991) *Schema Disruption*. Genetic algorithms and their applications: Proceedings of the Fourth International Conference, June 13-16 1991, (eds. R.K. Belew and L.B. Booker), University of California, San Diego, USA: Morgan Kaufmann. pp. 237-242.

Wang C.J. and Tsang E.P.K. (1991) *Solving constraint satisfaction problems using neural-networks*. Proceedings of the IEEE Second International Conference on Artificial Neural Networks. pp.

295-299.

Warwick T. and Tsang E.P.K. (1993a) *Using a Genetic Algorithm to Tackle the Processors Configuration Problem*. Technical Report CSM-190, 1993, University of Essex, Department of Computer Science, UK.

Warwick T. and Tsang E.P.K. (1994) *Using a Genetic Algorithm to Tackle the Processors Configuration Problem*. Proceedings of the ACM Symposium on Applied Computing (SAC), 1994, pp. 217-221.

Whitley D. (1988) *Applying Genetic Algorithms to Neural Network Learning*. Computer Science Department, Colorado State University, Fort Collins, Colorado 80523. To appear in: Proceedings of the Seventh Conference for the Study of Artificial Intelligence and Simulated Behavior, 1988, Sussex, UK: Pitman Publishing. pp. 137-144.

Whitley D. (1989) *The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 116-123.

Whitley D. and Ansonh T. (1989) *Optimizing Neural Networks Using Faster, More Accurate Genetic Search*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 391-397.

Whitley D. and Kauth J. (1989) *GENITOR: A different Genetic Algorithm*. Technical Report CS-88-101, Proceedings of the 1988 Rocky Mountain Conference on Artificial Intelligence, pp. 118-130.

Whitley D., Starkweather T. and Fuquay D'Ann. (1989) *Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator*. Genetic algorithms and their applications: Proceedings of the Third International Conference, June 4-7 1989, (ed. J.D. Schaffer), George Mason University, USA: Morgan Kaufmann. pp. 133-140.

Whitley D., Starkweather T. and Shaner D. (1991) *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. pp. 350-372.

Wilson S.W. (1987) *Hierarchical Credit Allocation in a Classifier System*. In *Genetic Algorithms and Simulated Annealing* (ed. L. Davis), London, UK: Pitman. pp. 104-115.

Definition Index

binary string S_{bin} 6
 building blocks 11

 car sequencing problem CarSP 67
 CSP 3

 diameter d_{max} 61

 epistasis 18

 GA-hard 22
 gray codes 15

 hamming cliffs 15

 implicit parallelism 11
 lower mean internode distance 65
 irregular network 61

 mean internode distance d_{avg} 62

 non-overlapping 7

 off-line 40
 on-line 40
 optimum graph 64

 PCSP 4
 PCSP(PCP) 65
 PCSP(CarSP) 76
 processor valency Δ 60
 processors configuration problem PCP 60

 regular network 60
 representation
 binary-coded 13
 real-coded 14

 schedule 68
 schema H 9
 defining length δ 9
 o-schema 22
 order 9
 search space 4
 string fitness 7
 string population $P(t)$ 6

 tightness
 ratio 117
 traveling salesman problem (TSP) 14

 upper bound n_{max} 62