Adaptive Search and Constraint Optimisation in Engineering Design

by

GEORGE ANGELOV BILCHEV

A thesis submitted to the University of Plymouth
in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

Plymouth Engineering Design Centre
School of Civil and Structural Engineering
Faculty of Technology

In collaboration with
British Aerospace
Rolls Royce and Associates

October 1996

Adaptive Search and Constraint Optimisation in Engineering Design

by

George Angelov Bilchev

ABSTRACT

The dissertation presents the investigation and development of novel adaptive computational techniques that provide a high level of performance when searching complex high-dimensional design spaces characterised by heavy non-linear constraint requirements. The objective is to develop a set of adaptive search engines that will allow the successful negotiation of such spaces to provide the design engineer with feasible high performance solutions.

Constraint optimisation currently presents a major problem to the engineering designer and many attempts to utilise adaptive search techniques whilst overcoming these problems are in evidence. The most widely used method (which is also the most general) is to incorporate the constraints in the objective function and then use methods for unconstrained search. The engineer must develop and adjust an appropriate penalty function. There is no general solution to this problem neither in classical numerical optimisation nor in evolutionary computation. Some recent theoretical evidence suggests that the problem can only be solved by incorporating *a priori* knowledge into the search engine.

Therefore, it becomes obvious that there is a need to classify constrained optimisation problems according to the degree of available or utilised knowledge and to develop search techniques applicable at each stage. The contribution of this thesis is to provide such a view of constrained optimisation, starting from problems that handle the constraints on the representation level, going through problems that have explicitly defined constraints (i.e., an easily computed closed form like a solvable equation), and ending with heavily constrained problems with implicitly defined constraints (incorporated into a single simulation model). At each stage we develop applicable adaptive search techniques that optimally exploit the degree of available *a priori* knowledge thus providing excellent quality of results and high performance. The proposed techniques are tested using both well known test beds and real world engineering design problems provided by industry.

Table of Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I am indebted to Ian Parmee, my director of studies. His critical advice has taught me the balance between careful research and exploratory investigation. I am particularly thankful for his patience and thrust, which has allowed me to pursue what has interested me.

I thank Andy Watson, Hary Vekeria and my colleagues from the Plymouth Engineering Design Centre for the many discussions we had concerning evolutionary algorithms. These discussions inspired many seminal ideas of my research.

I also thank British Aerospace plc. and Rolls Royce and Associates ltd. for providing my research with real world engineering design test problems.

I am grateful to my parents and sister for their unconditional support, knowing that doing so contributed greatly to my absence during the last years. They were strong enough to let me go easily, to believe in me, and to let slip away all those years during which we could have been geographically closer and undoubtedly driving each other crazy.

AUTHOR'S DECLARATION

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award.

This study was financed with the aid of a research assistantship from the EPSRC and carried out in the Plymouth Engineering Design Centre.

Relevant scientific seminars and conferences were regularly attended at which work was presented; external institutions (e.g. Isaac Newton Institute, Cambridge and Santa Fe Institute, New Mexico) were visited for consultation purposes and a number of papers prepared for publication.

Publications:

1. Constraint Handling for the Fault Coverage Code Generation Problem: An Inductive Evolutionary Approach, *Parallel Problem Solving from Nature IV*, Sep 96, Berlin, *LNCS* 1141, Springer, 1996, pp. 880-889
2. The Inductive Genetic Algorithm with Applications to the Fault Coverage Test Code Generation Problem, *Fourth European Congress on Intelligent Techniques and Soft Computing*, 2-5 September, 1996, Aachen, Germany, pp. 452-456
3. Optimization with an Ant Colony Search Model, Applications of AI for Technological and Business Processes, Vol. 2, ed. Martyn Polkinghorne, Univ. of Plymouth, ISBN 0 905227 573
4. Inductive Search, *First International Olympiad on Evolutionary Optimization held during the 1996 IEEE International Conference on Evolutionary Computation*, May 20-22, 1996, Nagoya, Japan, pp. 832-836
5. Learning the Next Dimension, *Procs. of the 1996 AISB Worskshop on Evolutionary Computing*, *LNCS* 1143, Springer, 1996, pp.162-174
6. Evolutionary Metaphors for the Bin-packing Problem, *5th Annual Conference of Evolutionary Programming*, 29 February - 02 March 1996, San Diego, USA
7. Constrained Optimization with an Ant Colony Search Model, *Adaptive Computing in Engineering Design and Control'96*, University of Plymouth, UK, pp. 145-151
8. Adaptive Search Strategies for Heavily Constrained Design Spaces, *Procs. of the 22nd International Conference on Computer-Aided Design*, Yalta, Ukraine, May 1995
9. The Ant Colony Metaphor for Searching Continuous Design Spaces, in (ed.) T. Fogarty, Evolutionary Computing, *Lecture Notes in Computer Science 993*, 1995, pp.25-39
10. Natural Self-organizing Systems, Internal report PEDC-02, Plymouth Engineering Design Centre, University of Plymouth, UK, March 1995

Signed .....
Date 24/09/97

CHAPTER 1

Introduction

1.1 Engineering Design and Optimality Issues

Engineering design does not seem to have a universally accepted definition. To some its important aspect is a broad planning function in which the general outline and form of a project are decided. To others it has an inventive connotation, describing the process of devising or selecting a solution to an engineering problem. The goal of the design process may range from providing a practical solution to a problem where none is previously known to improving on or replacing an existing design .

Design is usually referred to as the process of constructing a description of an artifact that satisfies a (possibly informal) *functional specification*, meets certain *performance criteria* and *resource limitations*, is realisable in a given *target technology*, and satisfies criteria such as *simplicity*, *testability*, *manufacturability*, *reusability*, etc.

Design is a complex human activity and different models of design have been proposed by various researchers. Fig. 1.1 illustrates the main features of a process based design model described by Gero [Gero, 92]. In this model, design is considered to consist of a number of distinct activities that are carried out between different design states. In many cases design

Design States

F = Set of design functions
B_e = Set of expected behaviours
B_s = Set of predicted behaviours
D = Design description
S = Design structure

Design Activities

F	→	B_e	Formulation
S	→	B_s	Analysis
B_e	→	S	Synthesis
B_s	→	B_e	Evaluation
S	→	B_e	Reformulation
S	→	D	Production

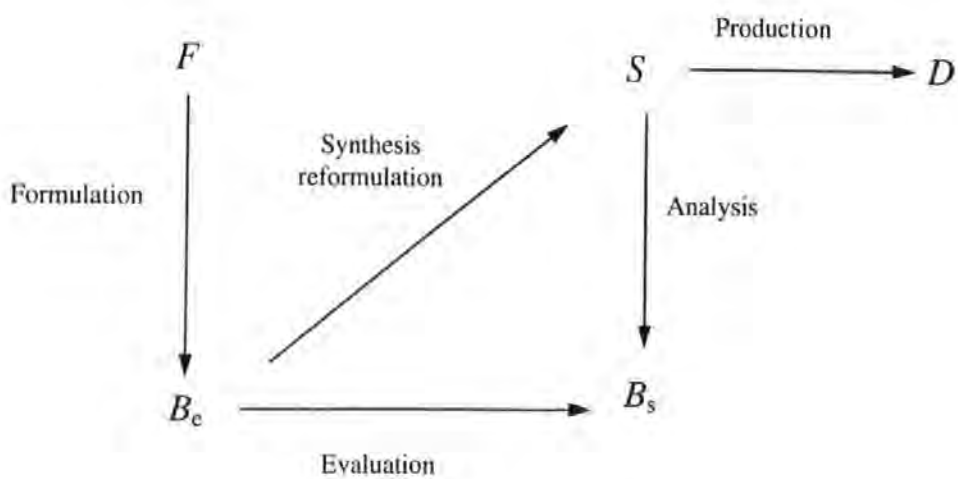


Fig. 1.1. Activities in design

can be viewed as a process of successive refinements where initially a design consists of a set of design functions (F) and a related set of expected behaviours (B_e). As the process of design progresses, the design structure becomes more clearly defined until finally a satisfactory structure is obtained (S). At the initial design stage, it is likely that the expected behaviour (B_e) and predicted behaviour (B_s) will be quite different, however as a design progresses predicted behaviour should approach expected behaviour.

In many design problems there are several possible alternative *design concepts*: for example, a girder for a highway bridge can be concrete or steel, and once the material is chosen, several approaches to using it are possible. Within these design concepts, there are variables which specify the dimensions, proportions, and other details of the item. Throughout this dissertation, we will adopt the point of view that a range of designs exist within a preselected design concept (usually implemented as a *simulation model*), and we will develop methods of choosing values to the quantities which prescribe the design. The *optimum* design aspect arises because we assume that these values are to be chosen in such a way that the design will be the one that satisfies all the limitations and restrictions (i.e., constraints) placed on it and is best in some sense. Our approach also considers those cases in which the major problem is to find any acceptable design in the presence of restrictions so severe or complicated that it is not clear how to proceed.

In other words, we will assume throughout the text that the engineering design problem has already been idealised, i.e., that a design concept has already been selected and that the design has been idealised into a mathematical (or computer simulation) model. While it may be said that this approach side-steps some of the most important questions in engineering, it should be emphasised that the main objective and contribution of this

dissertation is the development of a core of adaptive search design *tools* for decision support and not the design itself.

It should also be emphasised at this point that "a design" is simply a set of values for the design variables. Even if the design is patently absurd (e.g., negative areas) or inadequate in terms of function, it can still be called design. Clearly some designs are useful solutions to the design problem and others are not. If a design meets all the requirements placed on it, it will be called a *feasible* design or an *acceptable* design; the complement of the set of feasible designs will be called *infeasible* or *unacceptable* designs.

The design restrictions that must be satisfied in order to produce an acceptable design are collectively called *constraints*. The notion of constraint is central to design. Indeed, design has been often conceived as a process of expressing and exploring constraints. This certainly derives from the nature of design problems: something new must be created; human imagination is able to generate various possibilities; but this capacity and the alternatives it proposes must be managed by consideration of what is feasible. Constraints serve this purpose. They express relations among properties or variables of the proposed artifact and its environment or context [Maher, 89]. "Constraints are the rules, requirements, relations, conventions, and principles that define the context of designing" [Gross *et. al.*, 87].

We can identify two categories or kinds of constraints in engineering problems: side constraints and behaviour constraints. These categories are not necessarily definitive, as it may not always be easy to classify constraints in this way. However, since the classifications are mainly for convenience of communication, this is not a serious difficulty. A constraint that restricts the range of design variables for reasons other than the

direct consideration of performance is called a *side constraint.* A constraint that derives from behaviour requirements that are explicitly considered will be called a *behaviour constraint.* Another type of constraint which arises in some engineering problems is that of the discrete-valued design variable. In such cases the design variable is not to be selected from a continuous range of values but is permitted to take on only one of a discrete set of values. Such constraints can be very troublesome and it is usually the advantage of evolutionary population based search strategies that they can handle them quite efficiently.

Although the above classification of constraints is well established, it is difficult to classify problems according to it, since many real world problems include both side and behaviour constraints and discrete variables. Therefore, in this dissertation we will classify constrained problems by the way constraints are defined and expressed (sect. 1.5). The main reason behind this choice of taxonomy is that it greatly determines the choice of an appropriate optimisation tool as will be seen throughout the text.

Sometimes it is possible to incorporate all the restrictions imposed on the design into a single well-defined mathematical function (model). Each point from the domain of the function is a feasible design, leaving the main concern of searching for a "better" feasible solution. In other problems, restrictions are formulated explicitly and it is now up to the optimiser to combine them (possibly incorporating *a priori* preference information, i.e., degrees of "softness" and "hardness" of the constraints) in order to avoid infeasible designs. In many other engineering problems it is not even possible or practical to write explicit expressions for the constraints in terms of the design variables. For example, in a problem in which the stress is the final result of a finite difference computation or matrix inversion, the constraint cannot in general be put in an explicit form. More will be said later about this question. It suffices for now to state that the function which is limited by

the constraint must be a *computable* function of the design variables. This is not to say that the existence of explicit expressions for the behaviour functions is immaterial. On the contrary, this consideration often dictates the choice of optimisation method.

Of all feasible designs, some are "better" than others. If this is true, then there must be some quality that the better designs have more of than the less desirable ones do. If this quality can be expressed as a computable function of the design variables, we can consider optimising to obtain a "best" design. The function with respect to which the design is optimised is called the *objective* (also *cost* or *fitness*) *function*. The selection of an objective function can be one of the most important decisions in the whole optimum design process. In some situations an obvious objective function exists. Clearly *good* airplanes are not only light, but also have high payloads, long range, are economical to operate, inexpensive to buy, use reasonable runway length, etc. Care must be taken to optimise with respect to the objective function which most nearly reflects the *true* goals of the design problem. But experience shows that it is not always easy to decide whether a design characteristic should be associated with the objective function or with constraints. Let's consider the design of a disc brake, for which we assume the criterion is minimum stopping time. If we confront a vehicle project engineer with the problem he may well consider the stopping time for the vehicle as an acceptable design as long as it meets certain specifications. However, if we insist that the project engineer really think about how the performance characteristics affect the overall vehicle values, then he may begin to question the rigidity of the specifications. He may then decide that they all should be really in the objective function. Therefore, we say that we are in an area of intersection between the specification decision problem and the optimisation decision problem. This presents a dilemma: should an arbitrary decision be made on "hard" specifications for a design, based

on judgement, or should it be incorporated into the optimisation function as "soft" specifications.

1.2 On Reality and Models

It has been acknowledged that the necessary knowledge to model practical engineering systems and thereby capturing the breadth and depth of an engineer's expertise will be orders of magnitude larger than today's qualitative physics [Falkenhainer and Forbus, 88]. Engineering researchers have stressed that the complexity associated with modelling makes this task very difficult and argue that practical modelling systems are still some time away [Zienkiewicz and Zhu, 91]. As an example consider the protein-folding problem (i.e., given a linear sequence of amino acids, into what three-dimensional configuration will the sequence fold?). It is often stated that the protein-folding problem is NP-complete. It is crucial, however, to note that there are four worlds that come into play (fig. 1.2). Above the horizontal line are two real worlds; the world of bio-chemical phenomena and the computer world, where simulations are performed. These are worlds of atoms and electrons. Below the horizontal line are two formal models; a mathematical model of the bio-chemical phenomenon and a model of computation. In the formal models, representations are in bits. The mathematical model is an abstraction of the natural world while the model of computation is an abstraction of the computer world. The statement "protein-folding is NP-complete" co-mingles a real-world phenomenon with formal models. This is not an uncommon shortcut but if we are to make progress on a theory and practice of scientific and technological limits, it will be important to keep the distinction between reality and models clear.

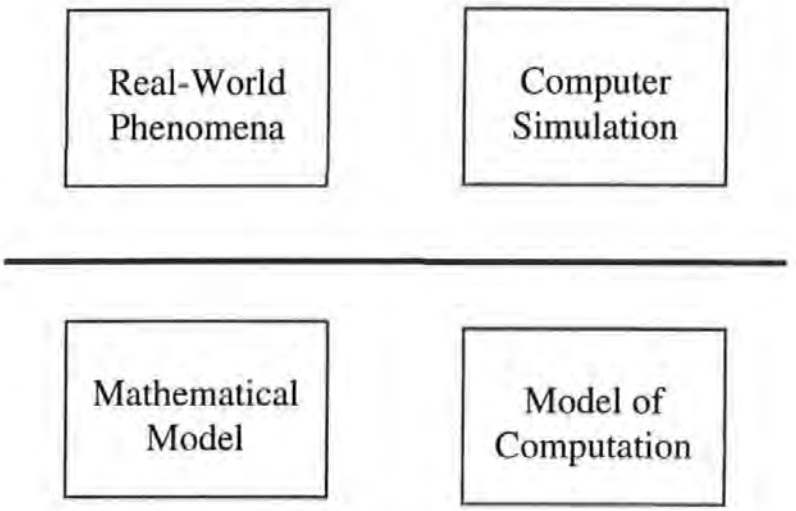


Fig. 1.2. Four worlds [Traub, 96].

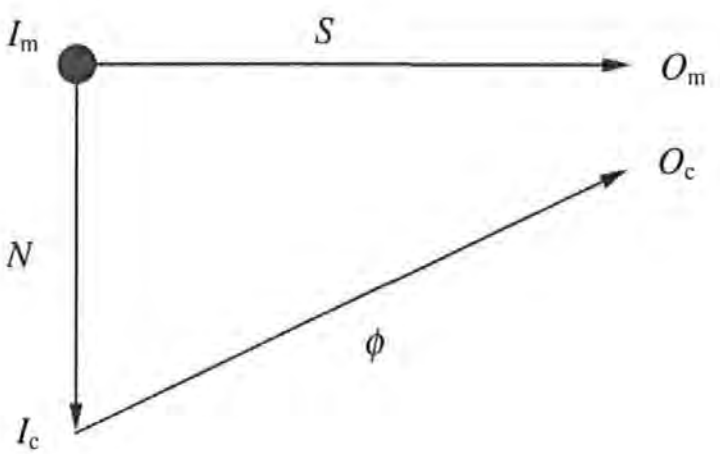


Fig. 1.3. Difference between the mathematical and computer simulated outputs

The mathematical problem to be solved is specified by the operator S (fig. 1.3) that maps the mathematical input, I_m, into the mathematical output O_m. This is very general since one can think of all computation as mapping inputs into outputs. Usually the mathematical input is a real multivariate function. Such a function cannot be input to a digital computer. Thus the function has to be replaced by a finite set of numbers, say, evaluating the function at a finite number of points. The operator N maps the mathematical input, I_m, into the computer input I_c. It is crucial that N is a many-to-one operator, i.e., knowing I_c does not give us I_m. Indeed, there are typically an infinite number of indistinguishable mathematical inputs corresponding to a computer input. A computer algorithm maps the computer input, I_c, into the computer output O_c. Note that $O_c \neq O_m$. Since N is many-to-one, we can't know which mathematical problem we are solving and therefore can, at best, solve the problem only approximately. Mathematically stated, N composed with ϕ does not commute with S.

This problem has been widely recognised in the Plymouth Engineering Design Centre [Parmee, 95a] where a far broader view of optimal engineering design has been established. The main objective is to identify an optimal design direction rather than optimal design solutions, which follows directly from the iterative nature of the design process (fig. 1.4) and the uncertainties in the models during the preliminary stages of the design. The identification of optimal direction relies upon a highly interactive process involving computer-based search tools, the development of which is the main contribution of this dissertation, engineering heuristics and design team decision making.

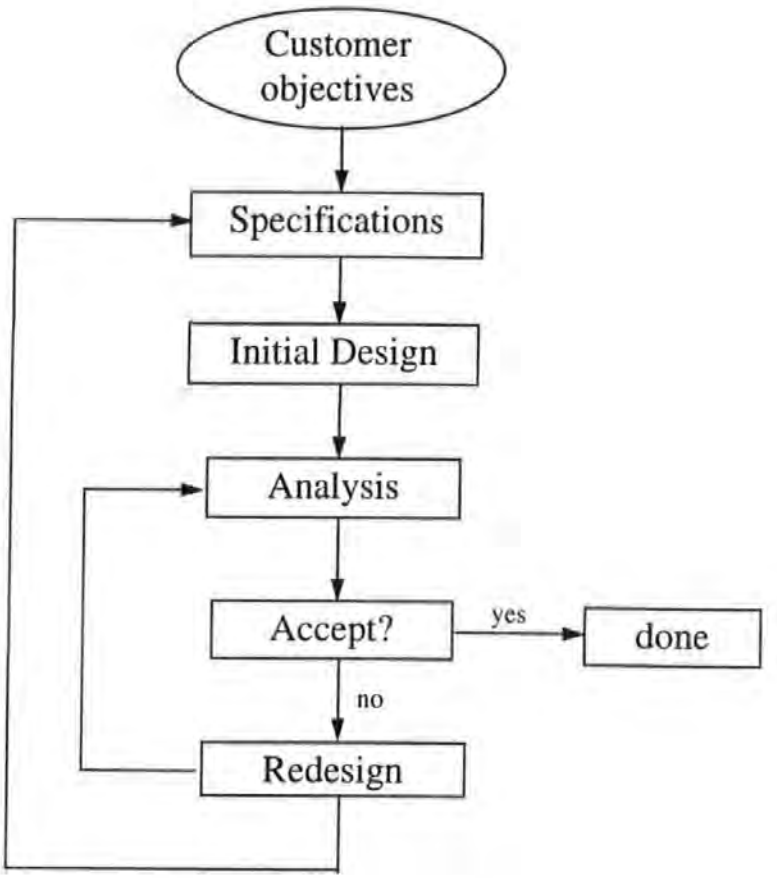


Fig. 1.4 The iterative model of design

1.3 The Role of Artificial Intelligence in Engineering Design

The late 1950s and the 1960s saw the development of the *search paradigm* within the field of Artificial Intelligence. Books such as "Computers and Thought" [Feigenbaum *et. al.*, 63], which appeared in 1963, were full of descriptions of various *weak methods* whose power lay in being able to view the solving of a particular kind of problem as a *search space*. In the late 1960s, the notion of *heuristic search* was developed, to account for the need to search large spaces effectively.

Nonetheless, most of the problems considered in those early days were what are now commonly called "toy problems". As the 1970s began, many practitioners in the field were concerned that the weak methods, though *general*, would never be *powerful* enough to solve real problems effectively; the search spaces would just be too large. Their main criticisms of the earlier work were that solving the toy examples required relatively little knowledge about the domain, and that the weak methods required knowledge to be used in very restrictive and often very weak ways. For example, in state space search, if knowledge about the domain is used, it must be expressed as either operators or evaluation functions, or else in the choice of the state space representation. The "weak method" critics took another approach, being primarily concerned with *acquiring* all the relevant knowledge into some usable form. Thus was born the "expert systems" paradigm.

During the 1970s, at the same time as many researchers were swinging to the "power" end of the "generality-power" trade-off curve in their explorations, others were striking a middle ground [Dixon, 86]. Some researchers, realising the limitations of the weak methods, began enriching the set of general building blocks out of which search algorithms could be configured.

Currently, AI contributes the notion of the design process as a search through a space of alternative designs; the synthesis tools are used to help generate new points in this space; the analysis tools are used to evaluate the consistency, correctness and quality of these points; the idea of search is used to guarantee that systematic progress is made in the use and re-use of the tools to generate new designs or design versions.

The price paid for search is efficiency, as the search space is generally quite large. Exhaustive search of the space is usually intractable; however, a search which focuses its attention on restricted but "promising" *subspaces* of the complete design space may trade away the guarantee of an optimal solution (provided by exhaustive search), in return for decrease in overall design time [Parmee and Denham, 94]. In this respect good *control heuristics* help. Control heuristics may either be domain-specific or domain-independent. "Spend much of the available design time optimising the component that is a bottleneck w.r.t. the most tightly restricted resource" is an example of a domain-independent heuristic, while "Spend much of the available design time optimising the datapath" is a domain-specific version of this heuristic that applies to certain situations in the microprocessor design domain. Designing appropriate control heuristics is a current state-of-the-art in optimal engineering design.

It is also worth noting a common misunderstanding which frequently arises between AI researchers who develop experimental Computer-Aided Design (CAD) tools, and traditional CAD tool developers in a particular design area who specialise in developing new design tools that will be usable in production mode in the near-term future. The CAD tool developers accuse the AI researchers of being too general. On the other hand, the AI researchers criticise the traditional CAD tool researchers of creating overly brittle systems. Confusion arises because these two types of researchers do not share the same research

goals. Traditional CAD tool developers seek to reduce the effort in creating *new designs*. Most AI researchers aim at reducing the effort in developing *new design tools*. Both research domains are worthy enterprises. The former goal requires the design tools to be powerful. The latter requires the methodology for constructing the tool to be general, and thus sometimes requires the design tool itself to be an instance of a general form rather than a custom-built tool.

In this dissertation we adopt an AI perspective of the engineering design process but most importantly we attempt to bring the gap between the two views closer by means of examples showing how to specialise generic adaptive search tools to particular engineering design applications. For this purpose a core of generic adaptive search engines is used in a variety of design contexts throughout the dissertation.

1.4 Previous Research

The *optimisation problem* is, in general, to find the optimum (maximum or minimum) value of a function in a given domain and to find the values of the variables where the optimum is reached in this domain. *Global optimisation* usually means to solve the optimisation problem in an unbounded area. *Local optimisation* means to solve the optimisation problem locally, that is, in the neighbourhood of a given point. Local optimisation has been investigated in depth; it has rich theory and many excellent numerical methods and recipes are available [Gill *et. al.*, 81][Press *et. al.*, 92]. Global optimisation, on the other hand, is a recent area which has been only partially researched [Törn *et. al.*, 88][Ratschek, 88]. Many theories have to be developed and many numerical experiments have to be performed before the area would be considered reasonably well developed. This research and development is, however, of the greatest importance since many real-world problems are global rather than local problems. Part of the contribution of

this dissertation is to also further enhance the development of adaptive global search methods and presents numerous experiments and real-world engineering design applications.

In order to allow a generalisation of the solution techniques to be presented and provide a uniform framework for discussion we now consider a standard form of problem statement. After making the appropriate engineering judgements and defining all the necessary functions and limitations, we state an optimisation problem as follows. Find the design X such that

$$\begin{aligned} & F(X) \rightarrow \min, \\ & s.t. \quad g_j(X) \leq 0, \end{aligned} \tag{1.1}$$

j=1,2,...,N. The problem is said to be stated in the design space. This form of problem is called a *mathematical programming problem*, or a *mathematical program*.

It is evident that the minimum may be a point where the constraints have no influence, as pictured in fig. 1.5. If F has continuous derivatives, the minimum is characterised as in the unconstrained case as:

$$\begin{aligned} & \nabla F = 0, \\ & J \equiv [\partial^2 F / \partial x_i \partial x_j] \text{ positive definite.} \end{aligned} \tag{1.2}$$

It is possible, however, for the situation depicted in fig 1.6 to exist in constrained problems. Here the minimum *admissible* design occurs at point P, where $\nabla F \neq 0$ but where one of

the $g_j(X) = 0$. If both g_j and F are differentiable, it is geometrically reasonable that a necessary condition for a minimum in this case is:

$$\begin{aligned} &\nabla F = \lambda \nabla g_j, \\ &\lambda < 0 \end{aligned} \tag{1.3}$$

where λ is some scalar. This requires that the contours of F be tangent to the constraint and both F and g_j increase in the same direction. This is not a sufficient condition as can be seen from fig 1.7 by examining point Q. Here ∇F and ∇g_j point in the same direction, but the contour bends away from the constraint. In this particular illustration the minimum may lie at points P or P'.

Fig. 1.8 shows yet another possibility. Here there are relative minima that are due to the form of the constraints, while those in fig 1.7 are due to the objective function. These distinctions are rather weak and defy rigorous definition.

We can sharpen the idea of necessary conditions for a relative minimum by stating an operational test which a proposed minimum must pass. This is called the *Kuhn-Tucker condition* [Fletcher, 87]. Roughly it consists of defining a cone expressed by the normals to all the active constraints at the point in question and then testing to see whether the gradient to the objective function is contained in the cone. However, the rigorous mathematical formulation of constrained optimisation relies on the assumption that the model is continuous and, moreover, doesn't take into consideration the uncertainties and

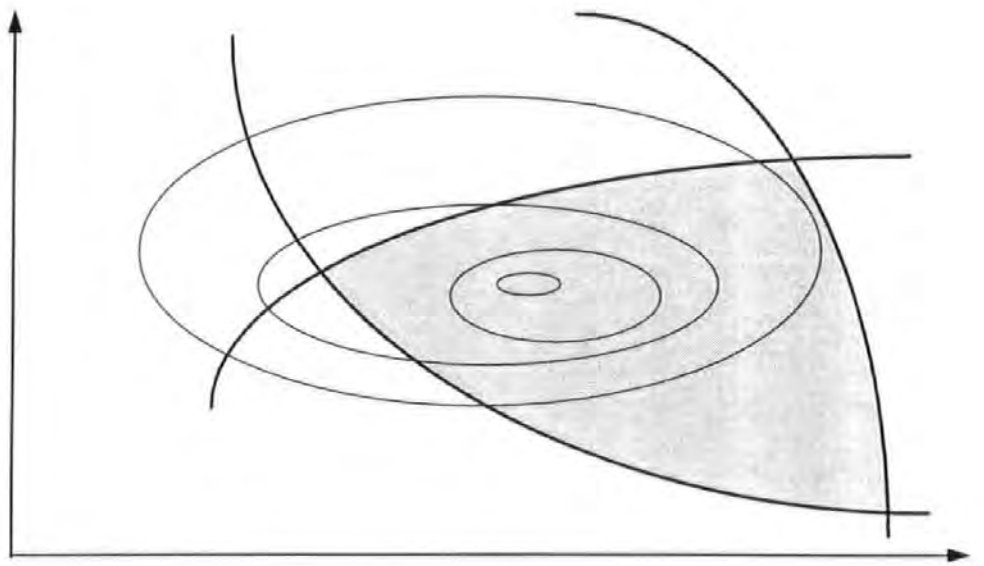


Fig. 1.5. Optimum defined by constrained optimisation. Scenario 1: The minimum is a point where constraints have no influence. The shaded region is the feasible region.

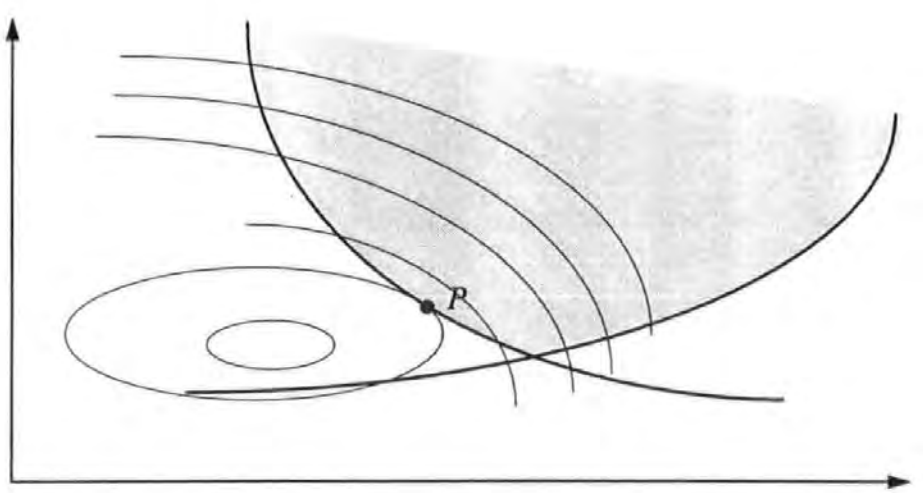


Fig 1.6. Optimum defined by constrained optimisation. Scenario 2: Active constraint at point P.

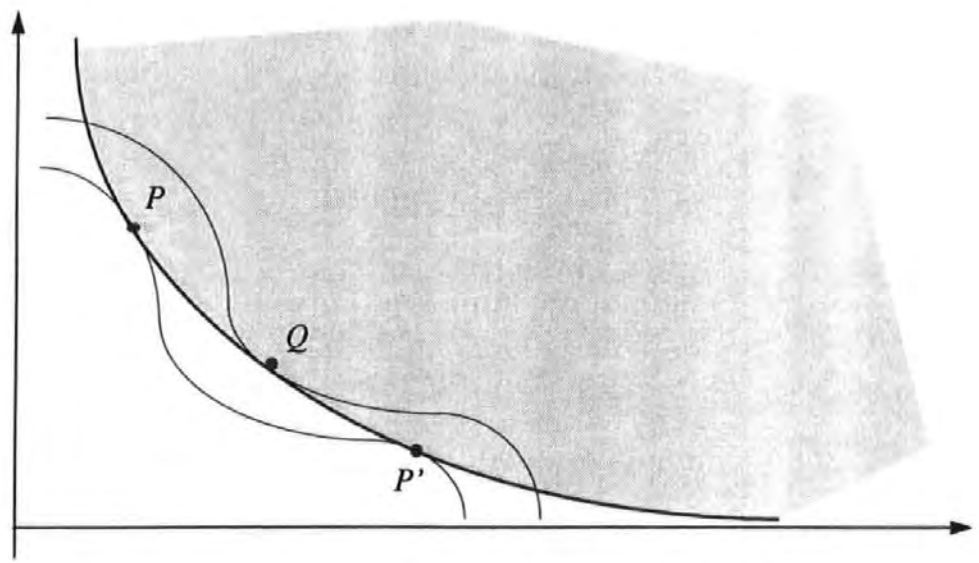


Fig 1.7. Optimum defined by constrained optimisation. Scenario 3: The gradients of the cost and constraint functions point in the same direction, but the contour of the cost function bends away from the constraint at point Q.

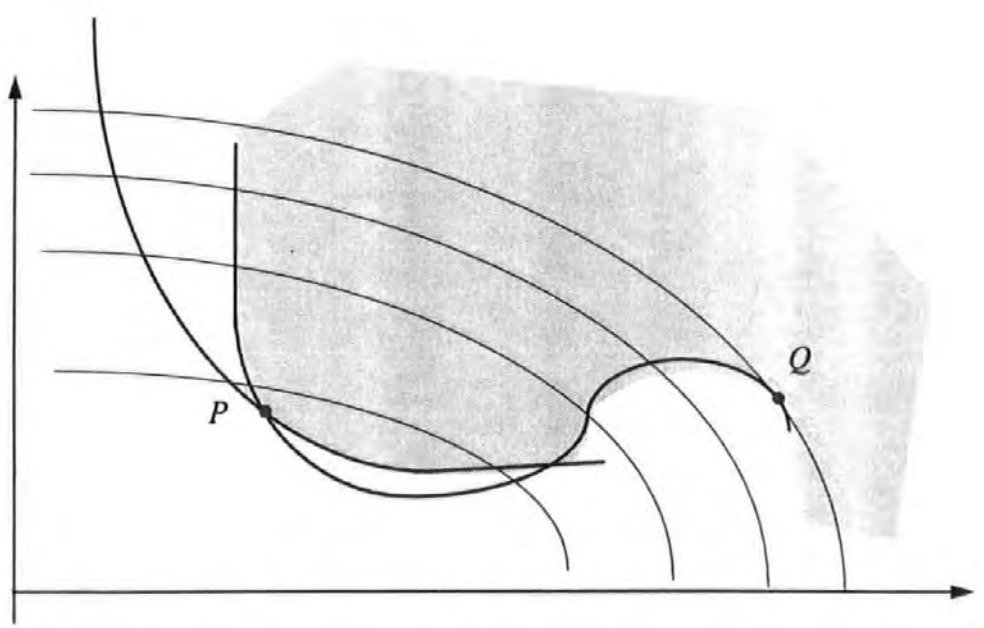


Fig 1.8. Optimum defined by constrained optimisation. Scenario 4: Intersection of constarints.

coarseness of the model itself. During the preliminary stages of the design process it is our major goal to find promising areas of the search space that will help to define an optimal design direction, rather than finding a solution that passes all known operational tests for optimality but w.r.t a very coarse and uncertain model [Parmee, 95a]. Therefore, at this stage the diversity of search is of paramount importance. Diverse adaptive sampling can be achieved by *population based search* methods and it is not a surprise that there have been numerous research efforts for application of evolutionary methods to real-world problems from a variety of domains: social systems, machine learning, operations research, ecology, engineering, immune systems, economics, management, etc.

Evolutionary computation techniques constitute an interesting category of heuristic search. Currently, the best known techniques in the class of evolutionary computation methods are genetic algorithms [Holland, 75], evolution strategies [Rechenberg, 64], evolutionary programming [Fogel *et. al.*, 66], and genetic programming [Koza, 92]. There are also many hybrid systems which incorporate various features of the above paradigms; however, the structure of any evolutionary computation algorithm is very much the same:

```
procedure evolutionary algorithm
begin
     t←0
     initialize P(t) // P(t) is the t^th population
     evaluate P(t)
     while (not termination-condition) do
     begin
          t←t+1
          select P(t) from P(t-1)
          alter P(t)
          evaluate P(t)
     end
end
```

There are numerous advantages of utilising evolutionary based search in the preliminary stages of the design process. If we use an evolutionary approach with penalty functions for

example, it is not essential for the penalty term to have any particular form, such as being unimodal or smooth, beyond having a fitness function that is easily evaluated. So it is not necessary to impose any continuity constraints on the penalty function, which is typically a very difficult task for the engineer in complex highly non-linear discontinuous parameter spaces. Moreover, the ill-conditioned problem usually associated with the numerical penalty function method does not exist for evolutionary based search because it uses ranking, i.e., it is easy to achieve total preference of the feasible over the infeasible solutions without changing the objective function [Powell and Skolnik, 93].

However, evolutionary algorithms have their own problems when used with penalty functions. In the context of highly constrained optimisation an infeasible solution with strong genotypic similarity to the optimal constrained solution is more useful in an intermediate population than a feasible solution with weaker genotypic affinity to the optimum. The problems that arise after introducing a penalty term can be summarised in the following way: an overzealous penalty rewards schema which quickly, but wastefully satisfy constraints. An over-tolerant penalty function will be unable to provide sufficient pressure to satisfy constraints and infeasible solutions will be highly fit. To a great extent these problems can be overcome by dynamic penalty functions. At the beginning the violated constraints are slightly penalised effectively warning the optimiser of the presence of boundaries while allowing their exploration. As the optimisation proceeds the violated constraints are severely penalised [Keane, 94]. The difficulties in the application of the dynamic penalty function is that the exact feasibility/infeasibility trade-off schedule cannot be effectively computed and is highly problem dependent [Richardson and Palmer, 89].

An attempt to overcome the above described difficulties is found in the *behavioural memory* approach [Schoenauer and Xanthakis, 93] where the constrained optimisation

problem is addressed by a multi-step process: (1) evolve an initial random population with some standard evolutionary search engine, the fitness function being related to the constraint satisfaction, and (2) take the final population resulting from this evolution and use it as an initial population of a secondary search with the objective function as fitness, which is overridden by zero fitness whenever the constraints are violated.

1.5 Dissertation Outline

The issues and contributions outlined in the previous sections are elaborated in the following chapters. Chapter 2 presents constraint handling at the representational level. Its purpose is to not only emphasise the crucial importance of the modelling phase, but to also develop and introduce through example applications the core adaptive search engines, namely the ant colony search model and the inductive search model. Themes of these adaptive search models will be present throughout the dissertation in various modifications at different levels of *a priori* knowledge utilisation. More specifically chapter 2 tackles the bin packing problem, showing how appropriate design of representation and search operators can avoid the difficulties associated with infeasible points. To achieve this, initially a distributed many-agent search model is developed. Next the basic ant colony search model is presented and extended to a hybrid search model and then applied to the bin packing problem. Handling constraints at the representation level is not only a virtue of combinatorial optimisation problems. Chapter 2 illustrates this by applying the technique for solving the protein-folding problem. Instead of utilising a grid based model of the free energy function of the protein at hand (which incurs problems of non-feasible points) it develops via geometric arguments a free energy model in which all points are feasible. This allows us to concentrate on the power of the adaptive search engine itself and chapter 2 achieves this goal by developing and introducing a fast high performance search engine called inductive search.

Chapter 3 deals with explicitly defined constraints. One of the highly efficient methods for handling such constraints is to derive the feasible region and apply a search engine only to the feasible space. Chapter 3 shows this by solving a real world problem of generating codes for test and monitoring systems. Provided that the feasible region is already explicitly derived, all we are left to do is apply an appropriate search engine. In order to show that the inductive search methodology introduced in chapter 2 is not intrinsically suited only for problems defined on continuous domains, we integrate it with a genetic algorithm and apply it to the test code generation problem.

Chapter 4 is a transition to problems with non-explicit (i.e., incorporated into a simulation model or black box) constraints. It treats increasingly complicated constraints where the feasible region can no longer be easily explicitly derived. In chapter 4 we extend the applicability of the ant colony search model to continuous spaces and show how to apply it to constraint optimisation problems.

Chapter 5 tackles the task of feasibility search (i.e., finding feasible design regions) for problems with non-explicitly defined constraints. With regards to the objectives of the engineering designer, chapter 5 mainly develops techniques to identify the scope and boundaries of the feasible region. Once this is done the search space can be reduced and the method from chapter 4 can be applied. In order to construct reliable methods we make use of recent theoretical developments on low discrepancy sequences and on analogies from immunology. Of particular attention is the minimisation of the number of calls to the cost function since in realistic problems this could be computationally very expensive.

To go even further in the difficulty of the constrained optimisation problem we reach the so called heavily constrained problems for which finding a feasible point is the major difficulty. Usually such problems have very small feasible regions which are disconnected and scattered. Therefore, the methods developed in the previous chapters will usually fail individually. This justifies the development of powerful hybrid search techniques whose primary goal is to find a feasible solution. This is the main concern in chapter 6, in which a hybrid ant colony and genetic algorithm are used to solve a preliminary airframe design problem. We also discuss constraint sensitivity issues important to the engineering designer and develop algorithms for their calculation/approximation.

Finally, chapter 7 provides conclusions and suggestions for future research.

CHAPTER 2

Evolutionary Search Methods that Model the Constraints in the Problem Representation

2.1. Distributed Many-Agent Search Model for Combinatorial Optimisation Problems (COPs)

2.1.1. Methodology

In this section we present a distributed many-agent search model which incorporates all the restrictions and constraints imposed on the problem by selecting an appropriate problem representation. In order to be able to do this the methodology requires a clear definition of the problems we wish to solve. Exact definitions of engineering problems are rarely encountered in practice with the exception of the well defined combinatorial optimisation problems (COPs). The exact formulation of COPs facilitates the problem representation constraint handling technique, thus concentrating all the attention towards the development of the adaptive search engine. The approach we will follow, often known as the *complex systems dynamics* approach [Weisbuch, 91], is to simplify as much as possible the components of the system, so as to take into account their large number.

We utilise the complex systems dynamics approach to tackle an ordering problem — loading of objects into minimal number of bins. A salient feature of the proposed search model is that the system is brought into an initial state corresponding to particular instance(s) of the problem to be solved and then it is allowed to evolve according to its own dynamics. The final state of this evolution is taken as a solution of the problem. The computation to be performed is contained in the dynamics of the systems which are determined by the nature of the local interactions between many simple elements. When designing the dynamical system care must be taken to ensure that all the states from the state space represent feasible solutions and that the motion operators preserve feasibility.

2.1.2. A Case Study: the Bin Packing Problem (BPP)

Falkenauer [Falkenauer, 94] describes the bin packing problem in the following way:

"The bin packing problem (BPP) is defined as follows: given a finite set O of numbers (the object sizes) and two constants C (the bin's capacity) and N (the number of bins), is it possible to pack all the objects into N bins, i.e., does there exist a partition of O into N or fewer subsets, such that the sum of the elements in any of the subsets doesn't exceed C ?

This NP-complete decision problem gives rise to the associated NP-hard optimisation problem [Garey and Johnson, 79]: what is the *best* packing, i.e., what is the *minimum* number of subsets in the above mentioned partition?

Being NP-hard, there is no known optimal algorithm for BPP running in polynomial time. However, Garey and Johnson cite simple heuristics which can be shown to be no worse (but also no better) than a rather small multiplying factor above the optimal number of

bins. The idea is straightforward: starting with one empty bin, take the objects one by one and for each of them first search the bins used so far for space large enough to accommodate it. If such a bin can be found, put the object there, if not, request a new bin. Putting the object into the first available bin found yields the First Fit (FF) heuristic. Searching for the most filled bin still having enough space to accommodate the object yields the Best Fit (BF), a seemingly better heuristic, which can, however, be shown to perform as well (as bad) as the FF, while being slower."

Other possible approaches for tackling the BPP are described in [Martello and Toth, 90],[Falkenauer, 94] and [Zulawinski, 95].

2.1.3. The Many-Agent Search Model

The many-agent (MA) search model consists of simple "agents" possessing only limited knowledge of how to interact with other agents. Each agent has several attributes: *capacity*, *strength*, number of successive *interaction failures*, and a list of *contained objects*. The loading problem considered here is one dimensional and has only one constraint — bins' *capacity*. The representation of each agent faithfully corresponds to a partially filled bin from a bin packing solution, and therefore, does not contain objects that violate the constraint (i.e., exceed the bins' capacity). The representation also takes care that none of the objects are shared between the agents and that each object belongs to a unique agent, thus ensuring that each state of the MA system (i.e., a vector consisting of the states of all individual agents) is a valid bin packing solution. The objects to be packed are characterised by one attribute, called *weight*. Input to the algorithm is a set of objects to be loaded and bin's capacity. The MA algorithm is defined as follows:

1. Load each object into an empty bin. Initialize the *strength* attribute of each box to be equal to its empty space, i.e., *capacity – object's weight*. It can be easily verified that the initial state is feasible (i.e., is a valid bin packing solution).

2. Select randomly two bins and initiate an inter bin operation (IBO). IBO is a feasibility preserving exchange of objects between two bins (see below).

   a) If IBO is successful (i.e., at least one object exchange has occurred) then update *strength* and the list of *contained objects* of both bins. If one of the bins is empty, then delete it from the set of agents. Otherwise, reset the successive *interaction failures* attribute of the stronger bin (i.e., the box with larger *strength* attribute) to zero.

   b) If IBO fails (i.e., no objects exchange has occurred), then increment the successive *interaction failures* attribute of the stronger bin. If it exceeds an interaction failure threshold τ (an integer usually between 1 and 5), then decrement the *strength* attribute of the stronger bin by a predefined constant ε (usually 2-5% from the bin's capacity).

3. If termination criteria are not satisfied, then goto 2. Otherwise exit with the current state as the final solution. The feasibility preserving of the IBO guarantees that the final state of the MA is a valid bin packing solution.

The inter bin operation (IBO) is a local interaction between two agents. It is used to implement the competitive drive in the evolution of the system and to maintain the feasibility of the problem representation (i.e., local exchanges of objects that violate the constraints are not allowed). During an IBO one of the agents is referred to as *strong*, and the other is referred to as *weak* (determined by the value of the *strength* attribute). The IBO algorithm works as follows:

1. Generate all k -tuples of objects in each bin, for some k=1 to C (complexity constant) as shown in fig. 2.1.

2. Add the empty space of the *weak* box to its k-tuples as shown in fig. 2.2., where the empty space is determined by:

$$empty\_space = capacity - \sum_i weight_i$$

3. If the two bins can exchange objects then find the best substitution (gain), i.e., an exchange of two k-tuples after which the empty space of the *strong* box increases at most (see fig. 2.3.) and the bin's capacity constraint is not violated.

The computational complexity of IBO is: $o\left(\binom{K}{C}\right)$, where *K=max(m,n), m=number of objects in the first bin, and n=number of objects in the second bin.* In the course of its evolution the MA search model reduces the number of agents (i.e., bins in the bin packing solution). The possibility that the *strength* attribute may differ from the empty space (step 2b of the main algorithm) is introduced as a mechanism for escaping local optima. Thus a strong agent which cannot complete a successful IBO (i.e., cannot gain empty space) becomes weaker and eventually other agents try to fill it in. If the *strength* attribute was always equal to the empty space then a bin which is emptier than most of the other bins, but containing a relatively "large" objects, would be more unlikely to find sufficient empty space and exchange it during an IBO in the course of evolution.

2.1.4. Experiments

In order to test our approach we use a distribution of objects which has been proved to be the worst case for the well known first fit in decreasing order (FFD) algorithm (a heuristic which sorts the objects in decreasing order and applies the FF algorithm). FFD has been shown to produce results better or equal to $\frac{11}{9}OPT(O)+3$, where $OPT(O)$ is the number of bins in the optimal solution [Baker, 85]. The distribution is defined as follows: $m=1,2,3,\ldots,\varepsilon=0.01$; $weight(obj_i)=\frac{1}{2}+\varepsilon$ (for $1\le i\le 6m$), $weight(obj_i)=\frac{1}{4}+2\varepsilon$ (for $6m\le i\le 12m$), $weight(obj_i)=\frac{1}{4}+\varepsilon$ (for $12m\le i\le 18m$), $weight(obj_i)=\frac{1}{4}-2\varepsilon$ (for $18m\le i\le 30m$).

Since our main objective in this chapter is to introduce the main search techniques that we have developed, our experiments do not aim at improving the best known techniques for solving the BPP, but to experimentally show the viability of our algorithms. Experiments are done for values of m from 1 to 5, i.e., for problem sizes ranging from 30 to 150

Fig.2.1. *k*-tuples of objects

Fig. 2.2. *k*-tuples with added empty space

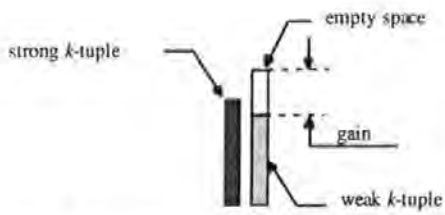


Fig. 2.3. Gain after an exchange of two *k*-tuples

Problem Size	30	60	90	120	150
optimal solution	9	18	27	36	45
FFD	11	22	33	44	55
MA	9.1	18.4	27.6	36.3	45.3

Table 2.1. Experiments with FFD worst case distribution.

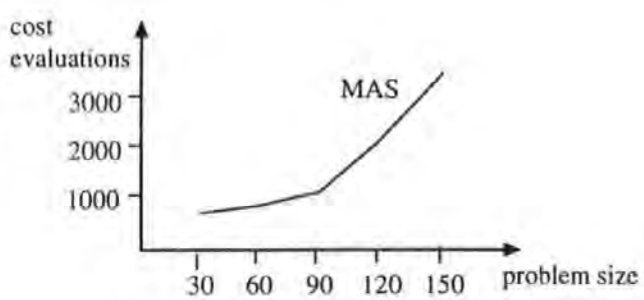


Fig. 2.4 Number of fitness evaluations as a function of the problem size.

objects. Empirical results, as summarised in table 2.1, show that the MA search model is complementary to the FFD heuristic and significantly outperforms it for the tested distribution. This complementarity is an important characteristic when designing hybrid search methods that employ different search strategies. This fact is used in the next section, where a hybrid ant colony search model will be introduced and developed. Fig 2.4. shows the number of local interactions as a function of the problem size. Results are averaged over ten independent runs.

To summarise, in this section we have developed a dynamical computational system for the bin packing problem. In our search model all the constraints associated with the BPP are handled by the problem representation; each point from the state space of the many-agent system is a valid bin packing solution. No global control and synchronisation is necessary since only local interaction rules are used to optimise the number of bins.

2.2. Ant Colony Search Model for COPs

Problems like the travelling salesman problem (TSP) and the bin packing problem (BPP) can be represented as a sequence of n items (n cities to be visited or n objects to be packed), where the actual order of the sequence uniquely determines a particular solution to the problem. Thus, in general, the feasible search space consists of all $n!$ permutations.

In this section we introduce the ant cycle (AC) algorithm [Colorni *et. al.*, 91]. It is defined as follows: The problem is represented as a connected graph the nodes of which are the n items. The edges are connections from one node to another and represent a data structure that stores the connectivity information in terms of the trail τ_{ij} left by the ants in the course of the algorithm's execution. Initially m ants are allowed to make a random cycle (a cycle is a permutation of n items). Then the cost function of the problem is evaluated for each

cycle and the graph connectivity information is updated by changing of the trail value (fig. 2.5a, 2.6a). The trail is defined by:

$$\tau_{ij}(t+n) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t, t+n) \tag{2.1}$$

where ρ is an evaporation constant ($0<\rho<1$), t is the time at the beginning of a cycle, and $t+n$ is the time at the beginning of the next cycle. $\Delta\tau_{ij}(t, t+n)$ is defined as follows:

$$\Delta\tau_{ij}(t, t+n) = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t, t+n) \tag{2.2}$$

where m is the number of ants, $\Delta\tau_{ij}^{k}(t, t+n) = G(f_k)$ if edge(i,j) is in the cycle of ant k, and is zero otherwise. In general the reward $G(f_k)$ is proportional to the fitness f_k of ant k.

Then the ants are allowed to make their next cycle but this time utilising the following procedure: select the first item randomly and then proceed to the next item by making a probabilistic decision defined in terms of the graph connectivity information (i.e., the trail value). The probability to visit item j when being at item i is:

$$P_{ij}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}(t)]^{\beta}}{\sum_{k \in allowed} [\tau_{ik}(t)]^{\alpha} \cdot [\eta_{ik}(t)]^{\beta}} & \text{if } j \in allowed \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where *allowed* is the set of items not visited for that particular cycle and η_{ij} is a local heuristic. α and β define a trade-off between the local heuristics and the ant colony search. Maintaining the tabu list *allowed* is of crucial importance to preserve feasibility of the solutions (i.e., an item is guaranteed to be visited at most once during each cycle). Next the

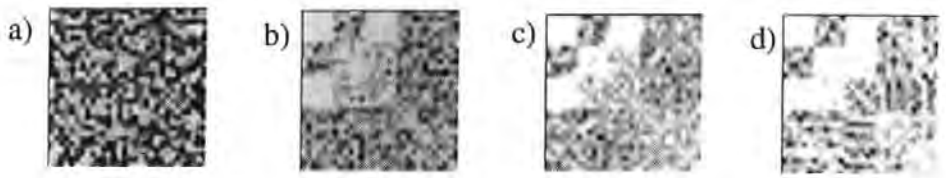


Fig: 2.5. Evolution of the connectivity pattern during different stages of the ant colony run: a) is the initial (random) pattern and d) is the pattern when the ant colony has converged. b) and c) represent intermediate patterns.

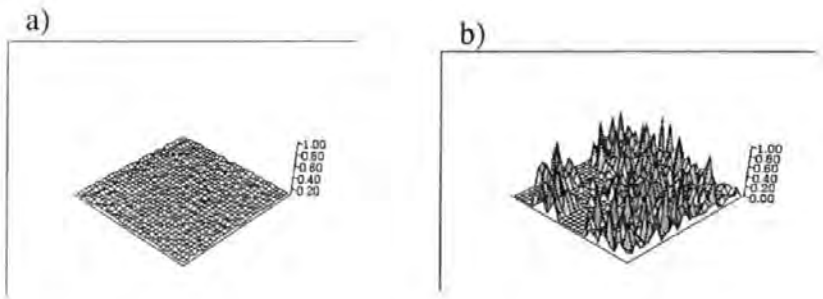


Fig. 2.6. Trail left by the ants: a) initial (random) trail at the beginning of the run; b) trail at the end of the run (i.e. when the ants' search has converged).

solutions are compared and trail is laid on the edges comprising the cycles proportionally to the ants' fitness. This alters the P_{ij} values so that on the next cycle the probability of repeating (part of) previous good cycles increases (fig. 2.5b,c,d, fig. 2.6b). This is quite reminiscent of *schemata propagation* in genetic algorithms where building blocks of high fitness pass from one agent to its offspring.

2.2.1. Extensions of the Ant Colony Metaphor

We begin this subsection with some initial definitions of distributed co-operative search [Clearwater *et. al.*, 92]. Co-operative search methods are based on modifying individual search strategies. A useful distinction is whether a method is *complete* or *incomplete*. Complete methods systematically examine states and are guaranteed to either eventually find a solution or terminate when no solution exists. By contrast, incomplete methods explore more opportunistically and may miss some states in the search space, hence they can never guarantee a solution does not exist. For parallel searches, a further issue is whether to split the search space among the agents. In the simplest case, each agent examines the entire search space. However this can mean a single state is examined by more than one agent during the search. This can be avoided by partitioning the search space into disjoint parts and assigning one to each agent. In this partitioned search, agents only examine states in their assigned part of the space thus avoiding unnecessary duplicate examination of states. Restricting each agent to examine a state at most once, as well as partitioning the search space so that a state is not examined by more than one agent, may improve performance somewhat, but far less than the enhancement achieved by co-operation [Clearwater *et. al.*, 92][Bilchev and Parmee, 95a][Bilchev, 96].

Next we generalise the ant colony search model to include search agents with different strategies. The completeness of the overall search depends in general on the completeness and complementarity of the individual strategies. The search space is not explicitly split

among the agents. However, the individual strategies are implicitly competing for a common resource — CPU time. The fitness measure of the competition is the success rate of each strategy measured from the beginning of the evolution. The overall evolution process resembles parallel competitive hypothesis testing from an evolving statistical sample (the current population). The hypotheses are the prior assumptions within which each individual search strategy has been designed to be effective. For example, consider two search strategies: a_1 and a_2 designed to be effective when applied to distributions of problem instances h_1 and h_2 respectively. When applying the ant colony to an instance of distribution h (for which we can assume that h may be either close to h_1 or h_2) then the generated search process could be viewed as hypothesis testing. The assumption is that if the current problem instance is generated from h_1 then a_1 will take control over a_2 (note that a_1 was initially designed to be effective on h_1). This process is somewhat analogous to the self-adaptation notion in evolutionary programming [Fogel *et. al.*, 95] and genetic algorithms [Bäck, 92].

For the BPP the extended ant colony (EAC) works as follows: Two new types of search ants are allowed — m agents using the MA strategy (section 2.1) and k agents utilising a genetic algorithm (GA) strategy [Bilchev, 96] (the paper is included in the Appendix). At each generation of the GA trail is laid on the tours of the k best solutions. To keep the grouping effect of the bin packing solutions trail is laid on all the edges connecting each of the objects from a particular bin. Then m ants are allowed to make a cycle and the MA search is applied on each cycle. Trail is laid (and superimposed) proportionally to the fitness of each cycle. In the current implementation of the extended ant colony search model both the GA and the MA strategy compete against each other by changing the ants' behaviour through the trail value. The feasibility of the solutions is guaranteed by a chain of feasibility preserving interfaces between the different search strategies, i.e., all

individual strategies use problem specific representation and feasibility preserving motion operators to search through the space of the feasible solutions. When one method passes a valid bin packing solution to another method the solution is "translated" into the representation language of the second search method.

2.2.2. Experiments

In order to test the ant colony search we make experiments with two distributions: the FFD worst case distribution and uniformly random distribution.

2.2.2.1 FFD Worst Case Distribution of Objects

The distribution is defined as in section 2.1.4. Experiments are done for values of m from 1 to 5, i.e., for problem sizes ranging from 30 to 150 objects. Empirical results are summarised in table 2.2, where results are averaged over ten independent runs and show that the extended ant colony search further improves on the performance of the many-agent system for this particular distribution of objects. The hypothesis which potentially explains this improvement is based on the assumption of complex adaptive interactions between several search strategies. In order to further test this hypothesis we do more experiments with uniform distribution of objects.

2.2.2.2. Uniform Distribution of Objects

Next we generate uniformly random sets of objects using the following parameters: *min. value: 0.05, max. value: 0.65, resolution: 300, and problem size: 50.* This time four algorithms are empirically compared: FFD, MA, GA, and EAC. Results are averaged over 10 independent runs for each technique and shown in fig. 2.7. In fig. 2.7 each graph is divided by vertical lines into sections. Each section corresponds to a particular number of

Problem Size	30	60	90	120	150
optimal solution	9	18	27	36	45
FFD	11	22	33	44	55
MA	9.1	18.4	27.6	36.3	45.3
EAC	9	18	27	36	45

Table 2.2. Experiments with FFD worst case distribution. For this particular distribution of object weights all our approaches outperform the FFD heuristic.

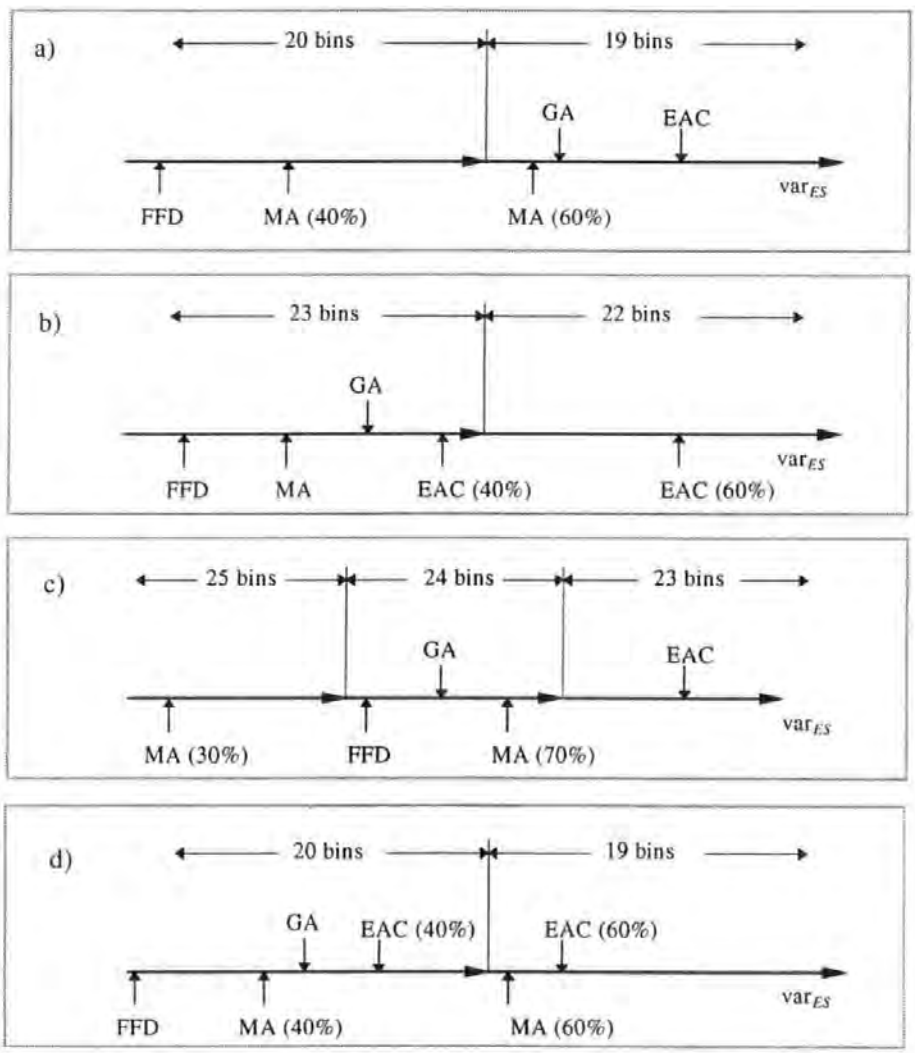


Fig. 2.7. Experiments with uniform distribution of objects. The extended ant colony search model (EAC) outperforms the quality of results of any of its comprising individual search strategies (FFD, GA, and MA). However, this is achieved by increasing the number of function calls from 2,500 for the MA and GA to 4000 for the EAC.

bins in the proposed solution. Better solutions are placed to the right of the graph. Within each section solutions are ranked in increasing order of the variance of their empty space. When a method produces results with different number of bins it is shown in more than one section of the graph.

The extended ant colony search model (EAC) generally outperforms any of its component individual search strategies at the expense of increased run time. Therefore, their performance diversity is very important for the overall performance of the hybrid model.

2.3 Inductive Search

2.3.1. The Protein-folding Problem

Handling constraints by appropriate selection of problem representation (i.e., model) is by no means limited to combinatorial optimisation problems. In this section we show how to utilise the approach for the prediction of the folded state of proteins, usually regarded as the *protein-folding* problem. Protein-folding phenomena present a daunting group of scientific challenges. Perhaps this is inevitable, since only a complex and diverse family of molecules could fulfil proteins' assigned roles in basic biological processes. The large and still rapidly growing literature on the subject of protein folding [Creighton, 84][Gierasch and King, 90][Nall and Dill, 91] chronicles many remarkable advances in both experiment and theory, yet this remains an open problem. Given an arbitrary but fully specified sequence of amino acids, we cannot yet predict the folding pathway of the corresponding polypeptide, the conformation of the final state, nor even verify in all cases whether the final state is one of lowest free energy or simply a metastable "trap" in the kinetic folding pathway.

One of the approaches to predict the protein-folding state is to model all the constraining forces imposed by the biochemical reactions among the residues. The resulting free energy model defines an energy landscape the global minima of which are believed to faithfully correspond to the folded state. However, it is also possible, due to external forces from the environment, for local minima to represent feasible metastable states. Abstracting away from the biochemical details of the problem, we will emphasise at this point that a set of (near global) minima from the energy landscape defines the feasible region of the search space. This requires the development and utilisation of a powerful adaptive search engine that is able to find the set of feasible states.

2.3.2. The Energy Landscape Model

The model which we have utilised is defined by Stilliger [Stillinger *et. al.*, 93] as follows: "[The] model incorporates only two "amino acid" types, to be denoted by A and B, in place of the 20 that occur naturally. They will be linked together by rigid unit-length bonds to form linear unoriented polymers that reside in two dimensions. As fig. 2.8 illustrates, the configuration of any n-mer is specified by the n-2 angles of bend $\theta_2,...,\theta_{n-1}$ at each of the nonterminal residues. We adhere to the conventions that $-\pi \leq \theta < \pi$, that θ_i =0 corresponds to linearity of successive bonds, and that positive angles indicate counterclockwise rotation.

We postulate that two kinds of interactions compose the intramolecular potential energy for each molecule: backbone bond potentials (V_1) and nonbonded interactions (V_2). The former will be independent of the A, B sequence, while the latter will vary with that sequence and will receive a contribution from each pair of residues not directly attached by a backbone bond.

Residue species along the backbone can be conveniently encoded by a set of binary variables $\xi_1,\ldots,\xi_n$. If $\xi_i=1$, the ith residue is A; if $\xi_i=-1$, it is B. The intramolecular potential-energy function Φ thus can be expressed as follows for any n-mer:

$$\Phi = \sum_{i=2}^{n-1} V_1(\theta_i) + \sum_{i=1}^{n-2} \sum_{j=i+2}^{n} V_2(r_{ij}, \xi_i, \xi_j) \tag{2.4}$$

The distances r_{ij} can be written as functions of the intervening angles (recall that backbone bonds have unit length):

$$r_{ij} = \left\{ \left[\sum_{k=i+1}^{j-1} \cos\left(\sum_{l=i+1}^{k} \theta_l \right) \right]^2 + \left[\sum_{k=i+1}^{j-1} \sin\left(\sum_{l=i+1}^{k} \theta_l \right) \right]^2 \right\}^{\frac{1}{2}} \tag{2.5}$$

Our model assigns a simple trigonometric form to V_1:

$$V_1(\theta_i) = \frac{1}{4}(1 - \cos\theta_i) \tag{2.6}$$

The nonbonded interactions V_2 have a species-dependent Lennard-Jones 12,6 form [Creighton, 84]:

$$V_2(r_{ij}, \xi_i, \xi_j) = 4\left[r_{ij}^{-12} - C(\xi_i, \xi_j) r_{ij}^{-6} \right], \tag{2.7}$$

$$C(\xi_i, \xi_j) = \frac{1}{8}(1 + \xi_i + \xi_j + 5\xi_i\xi_j) \tag{2.8}$$

If only V_1 mattered, successive bonds would tend toward linearity ($\theta_i = 0$). The coefficient $C(\xi_i, \xi_j)$ is +1 for an *AA* pair, $+\frac{1}{2}$ for a *BB* pair, and $-\frac{1}{2}$ for an *AB* pair. Consequently the first of these pairs may be regarded as strongly interacting, the second as weakly interacting, and the third as weakly repelling. This diversity mimics in a simple way that of real amino-acid residues, which vary in size, polarity, and degree of hydrophobicity. In fact, it can be assumed that *A* and *B* behave respectively as hydrophobic and hydrophilic residues. The interplay between the backbone bend interaction that tends to produce linear structures, and the various combinations of attractive and repulsive nonbonded pair interactions, generates a wide range of ground-state geometries. It is in this respect that our model remains faithful to the character of real proteins." A C-code implementation of the model is included in the Appendix.

2.3.3. The Inductive Search Engine

2.3.3.1. Introduction

In this section we develop a novel powerful search engine. Our research is strongly motivated from the fact that currently evolutionary optimisers display rather slow convergence rates and poorer quality of solutions as compared to their numerical local optimisation counterparts searching in the correct neighbourhood. This research commenced from our beliefs that the fundamental principles of genetic algorithms are quite relevant to the problem of optimisation of real-valued functions if properly utilised.

Our algorithm is based on the assumption that an approximation of the desired solution can be effectively *constructed* from a limited sample of the search space. The idea is generally borrowed from genetic algorithms and the corresponding *building block hypothesis* [Goldberg, 89], but is utilised in a more direct way. We also view the global optimisation problem as an existence of short (inductive) rules that can effectively build the solution

from a limited sample. If for a particular problem instance such rules do not exist, the global optimisation task is not tractable. On the other hand, if such rules exist, the current state of the art is how to find them.

2.3.3.2. The Framework and the Rationales Behind its Design

A novel feature in our search algorithm is that we do not constrain the sampling procedure to work only in the original search space, but rather divide the problem into a sequence of subproblems and allow sampling in each of the newly defined subspaces. In order to do that our approach requires a computational model of the cost function, which is usually available.

Currently, it is widely accepted that dimensionality is an important characteristic which determines to a great extent the tractability of the search problem. Therefore, many test beds include scaleable functions which are defined for any dimension N:

$$f_1(x_1), f_2(x_1,x_2),\ldots, f_N(x_1,\ldots,x_N),\ldots \tag{2.9}$$

Each new dimension is *derived* from the previous by adding a new variable and defining its interactions with the other variables. Therefore, it is natural to make the *a priori* assumption that in our computer models the set of local optima of the "next dimension" can be *derived* from the set of local optima of the current dimension (fig. 2.9). The above assumption justifies the following general search framework:

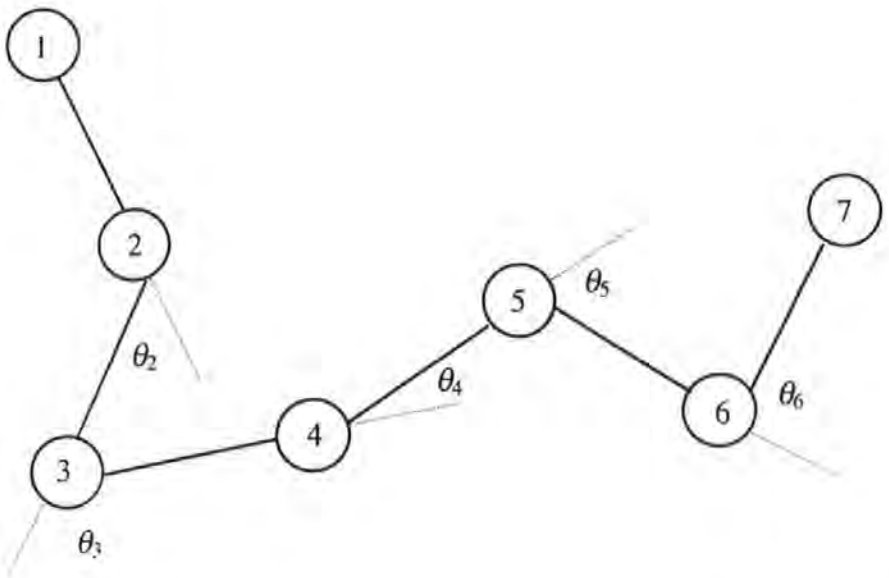


Fig 2.8. A schematic diagram of a generic 7-mer, with serially numbered residues, and backbone bend angles.

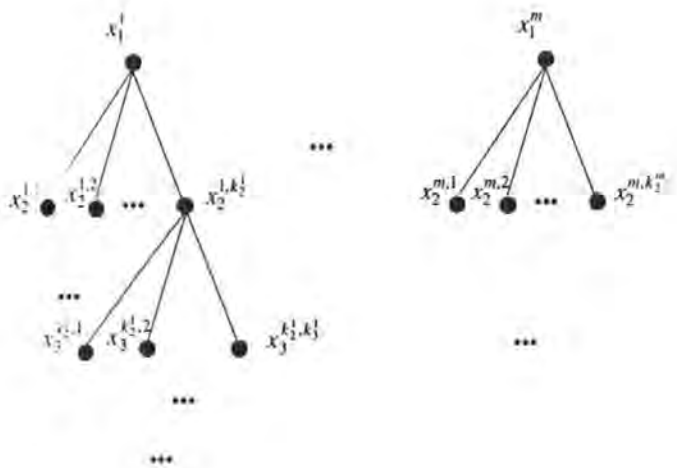


Fig. 2.9. At each (horizontal) level the graph is read as follows: The lower index shows the current dimension. The upper index is a pair, the first number of which shows the parent from the previous dimension (i.e., upper level) and the second number enumerates the set of local minima of the current level. Assumption: the set of local optima of dimension N can be *derived* from the set of local optima of dimension N–1. The forest structure also shows how the "curse of dimensionality" phenomenon emerges.

1. Find the set of local minima of the 1-dimensional version of the cost function.

2. Initialize D=1. D is the current dimension.

3. Get the best M minima and form a population of M D-dimensional points. Increase the current dimension by one: D=D+1

4. For each member of the population form a D-dimensional version of the cost function where the first D-1 variables are fixed equal to those from the selected member (ref. figs. 2.10, 2.11). Use 1-dimensional global optimization method to find a set of new local minima. For the best solutions apply a hill climber to locally improve on the results and place them in a pool of offspring. The members from the pool will be points from a D-dimensional space.

5. If D=N exit, otherwise goto 3

2.3.3.3. Implementation Details

There are many efficient (under certain assumptions) one dimensional search algorithms which can be utilised. Examples include algorithms based on the Wiener process, statistical-informational methods, interval methods, etc. [Törn and Zilinskas, 88][Ratschek and Rokne, 88]. In this section we have implemented a simple but efficient (under certain smoothness assumptions) one dimensional search algorithm. It utilises Brent's quadratic approximation method [Press *et. al.*, 92] and works as follows:

Step 1 Initialize a population of search intervals, i.e. I={[a,b]}.

Step 2 For the largest interval $\in$ I make a quadratic approximation, i.e., c=brent(a,(a+b)/2,b).

Step 3 The selected interval from step 2 is divided into three subintervals determined by the two inside points: c and (a+b)/2 and the subintervals are inserted into I.

Step 4 If the stopping criterion is satisfied then exit; otherwise goto 2.

Various stopping criteria are possible. Examples include size of the largest interval, number of quadratic approximation calls, value to reach. The implemented criterion is

number of quadratic approximation calls. The choice to explore the largest interval in step 2 is well justified by the desire to minimise the maximum risk of missing a solution.

The local hill climber is implemented as a local search algorithm using only values of the cost function. In our experiments we have implemented the hill climbing aspects of dynamic hill climbing (DHC) [Yuret, 94].

2.3.3.4. Experiments

In order to better explain the behaviour of our algorithm we first run through its basic steps using the following test function (Langerman's test function):

$$f(\bar{x}) = -\sum_{i=1}^{m} c_i \left(e^{-\frac{1}{\pi}\|\bar{x}-A(i)\|^2} \cdot \cos\left(\pi \cdot \|\bar{x}-A(i)\|^2\right) \right), \quad m = 5 \tag{2.10}$$

Initially, the global optimisation method is used to find the set of local minima of the 1-dimensional version of the test function (fig. 2.10):

$$\begin{aligned} x_1^1 &= 2.197, & f(x_1^1) &= -0.908 \\ x_1^2 &= 6.628, & f(x_1^2) &= -1.276 \\ x_1^3 &= 8.184, & f(x_1^3) &= -2.709 \\ x_1^4 &= 9.450, & f(x_1^4) &= -2.579 \end{aligned}$$

Then if we fix $x_1^* = x_1^3$ we can find the set of local minima of $f_2(x_1^*, x_2)$ (fig. 2.11):

$$\begin{aligned} x_2^1 &= 0.489, & f(x_1^*, x_2^1) &= -0.396 \\ x_2^2 &= 6.746, & f(x_1^*, x_2^2) &= -0.432 \\ x_2^3 &= 7.237, & f(x_1^*, x_2^3) &= -0.425 \\ x_2^4 &= 8.995, & f(x_1^*, x_2^4) &= -2.389 \end{aligned}$$

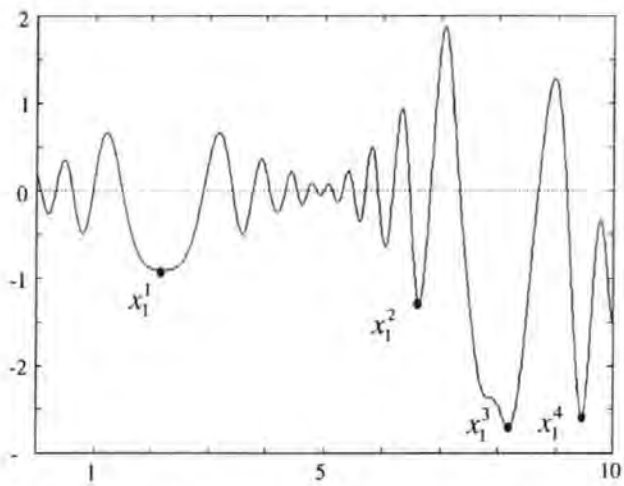


Fig. 2.10. Langerman's function for D=1.

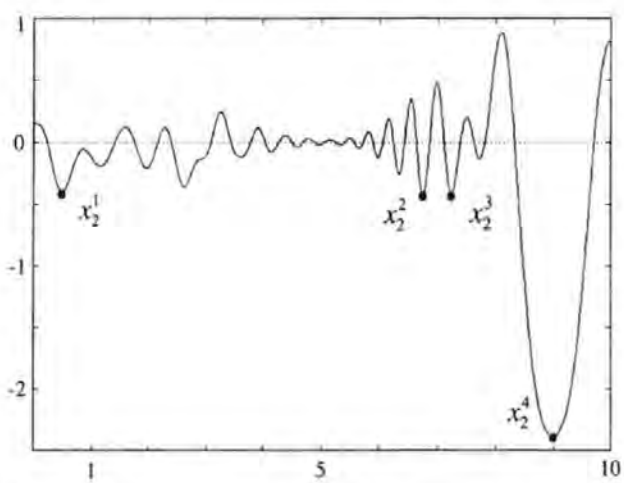


Fig. 2.11. One dimensional version of Langerman's function for D=2, where $x_1^* = x_1^3$ (fig. 2.10).

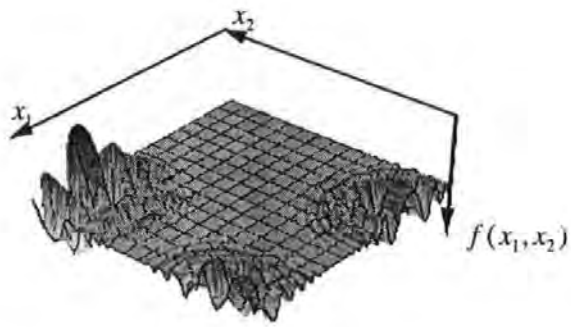


Fig. 2.12. Langerman's function for D=2.

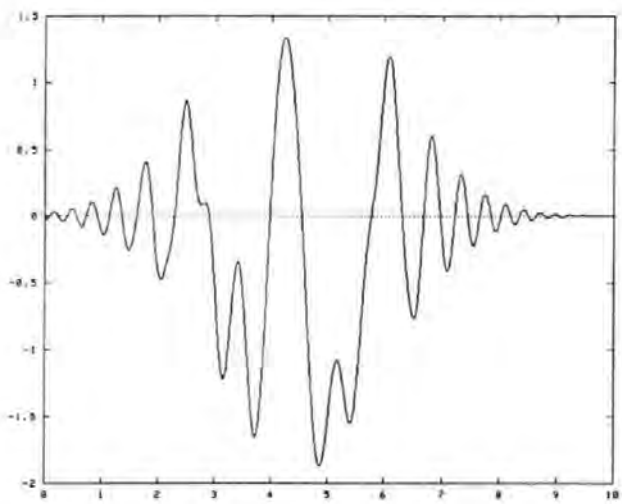


Fig. 2.13. One dimensional version of Langerman's function for D=3, where $x_1^* = x_1^3$ (fig. 2.10) and $x_2^* = x_2^4$ (fig. 2.11).

Next, by using a hill climber starting from $f(x_1^*, x_2^*)$, where $x_2^* = x_2^4$, we can update the values of x_1^* and x_2^* to 8.047 and 8.985 respectively. An overview of $f_2(x_1, x_2)$ is given in fig. 2.12. The new values are used in the next iteration, where we seek for the set of local optima of $f(x_1^*, x_2^*, x_3)$ (fig. 2.13). The process is further iterated.

The proposed inductive search algorithm was awarded the first prize at the First International Contest on evolutionary optimisation held during the 1996 IEEE conference on Evolutionary Computation. The proposed test bed for the competition consists of the following functions which are to be minimised for various dimensions:

TEST FUNCTION	RANGE	N
The sphere model: $f_1(\bar{x}) = \sum_{i=1}^{N} (x_i - 1)^2$	$x_i \in [-5, 5]$	5 10
Griewank's function: $f_G(\bar{x}) = \frac{1}{4000} \sum_{i=1}^{N} (x_i - 100)^2 - \prod_{i=1}^{N} \cos\left(\frac{x_i - 100}{\sqrt{i}}\right) + 1$	$x_i \in [-600, 600]$	5 10
Shekel's foxholes: $f_S(\bar{x}) = -\sum_{i=1}^{m} \frac{1}{\|\bar{x} - A(i)\|^2 + c_i}, \quad m = 30$	$x_i \in [0, 10]$	5 10
Michalewicz's function: $f_M(\bar{x}) = -\sum_{i=1}^{N} \sin(x_i) \cdot \sin^{2m}\left(\frac{i \cdot x_i^2}{\pi}\right), \quad m = 10$	$x_i \in [0, \pi]$	5 10
Langerman's function: $f_L(\bar{x}) = -\sum_{i=1}^{m} c_i \left(e^{-\frac{1}{\pi}\|\bar{x} - A(i)\|^2} \cdot \cos\left(\pi \cdot \|\bar{x} - A(i)\|^2\right) \right), \quad m = 5$	$x_i \in [0, 10]$	5 10

The results achieved by the inductive search are summarised in table 2.3. Compared with the other participating algorithms our inductive search exhibits an extremely small number of cost function calls needed to construct the global minimum of the proposed test functions. Our explanation of these results is that the inductive search succeeds in utilising domain specific knowledge.

FUNCTION	Dimension	Cost function calls	Best value
The sphere model	N = 5	20	3.88e-15
	N = 10	40	7.10e-15
Griewank's function	N = 5	41	7.99e-6
	N = 10	79	1.31e-6
Shekel's foxholes	N = 5	74	-10.327
	N = 10	120	-10.101
Michalewicz's function	N = 5	120	-4.6876
	N = 10	501	-9.6600
Langerman's function	N = 5	176	-1.499
	N = 10	372	-1.499

Table 2.3: Performance indexes.

Now we proceed with the application of the inductive search approach to the protein-folding problem. For any number n of residues, and for any given sequence of those n residues specified by $\xi_1,...,\xi_n$, the potential-energy function Φ is precisely defined, and in principle can be minimised with respect to the conformational angles $\theta_2,...,\theta_{n-1}$. In practice this is easy for small n (i.e., 3,4, and 5), but becomes increasingly tedious and demanding as n increases. In previous research [Stillinger *et. al.*, 93] an exhaustive search is used to generate a database of ground states for small n. The database is shown in table 2.4. The simple case of trimers (n=3) provides an introductory illustration. The six distinct molecules are *AAA*, *AAB*, *ABA*, *ABB*, *BAB*, and *BBB*. Each has only a single bend degree of freedom θ_2. Furthermore, the potential energy as defined by our model, depends only on the species of the terminal residues, and not on that of the central residue. Consequently there are just three distinct cases to consider: *AXA*, *AXB*, and *BXB*.

AXB and *BXB* trimers are linear in their ground states. That is certainly expected for *AXB*, where the terminal residues repel each other at all separations. Even though modest terminal residue attraction exists for *BXB*, the bend potential energy is sufficiently costly that the possibility of an absolutely stable bent shape is eliminated. Only when both terminals are *A* is the nonbonded interaction sufficiently attractive to generate a bent ground state; θ_2 is approximately $\pm 111.4°$ in this nonlinear structure. However, the *AXA* molecules retain the linear form as a metastable (relative) Φ minimum; this is the first appearance of multiple minimum problem that magnifies dramatically in severity as the molecules increase in residue number.

Before applying our algorithm we have selected the following parameters: at each dimension only the best three solutions from the population are accepted and for each of them only five calls to Brent's approximation routine are made. This choice of parameters

significantly reduces the overall number of cost function calls while at the same time preserves the quality of solution. The inductive search is capable of finding all but one ground state. Results are shown in table 2.5. The case in which the inductive search fails is *ABBBA*, where a metastable ground state is predicted (i.e., the linear structure). It is interesting to note why the inductive search fails in this case. All subsequences that compose *ABBBA* have their global minima in their linear structure (i.e., *ABB*, *BBB*, *ABBB*). That is to say that the one dimensional (*ABB*) and two dimensional (*ABBB*) versions of the cost function do not provide the necessary information to find the global optimum of the three dimensional (*ABBBA*) version. The information they provide is only local and therefore, the inductive search finds a local optimum.

Although the inductive search is capable of efficiently optimizing our free energy protein-folding model, it is by no means the best algorithm for the problem. As pointed out by Prof. Keane, the problem can be even more efficiently solved by a combination of dynamic hill climbing followed by a local climber such as the method of Hooke and Jeves. For example, such a combination is capable of solving the *ABBBA* instance of the protein-folding problem in 543+202 number of cost function calls.

Molecule	Φ	θ_2/π	θ_3/π	θ_4/π
AAA	-0.65821	0.61866		
AAB	0.03223	0.00000		
ABA	-0.65821	0.61866		
ABB	0.03223	0.00000		
BAB	-0.03027	0.00000		
BBB	-0.03027	0.00000		
AAAA	-1.67633	0.61839	0.33920	
AAAB	-0.58527	0.61759	-0.05130	
AABA	-1.45098	0.33270	0.62180	
AABB	0.06720	0.00000	0.00000	
ABAB	-0.64938	0.61767	-0.06670	
ABBA	-0.03617	0.47690	0.47690	
ABBB	0.00470	0.00000	0.00000	
BAAB	0.06172	0.00000	0.00000	
BABB	-0.00078	0.00000	0.00000	
BBBB	-0.13974	0.55828	0.35180	
AAAAA	-2.84828	0.33597	0.62022	0.04543
AAAAB	-1.58944	0.61898	0.33748	-0.06894
AAABA	-2.44493	0.29723	0.33306	0.62176
AAABB	-0.54688	0.61756	-0.05373	-0.00168
AABAA	-2.53170	0.32943	0.62354	0.04551
AABAB	-1.34774	0.33269	0.62133	-0.54574
AABBA	-0.92662	0.16722	0.48228	0.47327
AABBB	0.04017	0.00000	0.00000	0.00000
ABAAB	-1.37647	0.62222	0.33110	-0.06303
ABABA	-2.22020	0.61900	0.04739	0.61900
ABABB	-0.61680	0.61765	-0.07104	-0.00224
ABBAB	-0.00565	0.47880	0.47341	-0.14184
ABBBA	-0.39804	0.24576	0.55551	0.24576
ABBBB	-0.06596	0.05489	-0.34237	-0.56178
BAAAB	-0.52108	0.03924	-0.61671	0.03924
BAABB	0.09621	0.00000	0.00000	0.00000
BABAB	-0.64803	0.05328	-0.61682	0.05328
BABBB	-0.18266	0.56920	0.33574	0.26659
BBABB	-0.24020	0.31773	0.57642	0.09738
BBBBB	-0.45266	0.34345	0.56501	0.09318

Table 2.4. Ground-state properties of toy-model polypeptides. Angles θ_i are measured in radians. Molecules are listed in alphabetical order for each number of residues, and, in case of sequences differing only by reversal, only the first in alphabetical order appears.

Molecule	Φ	Cost function calls
AAA	-0.65821	636
AAB	0.03223	688
ABA	-0.65821	636
ABB	0.03223	688
BAB	-0.03027	660
BBB	-0.03027	660
AAAA	-1.67633	1937
AAAB	-0.58527	1617
AABA	-1.45098	2391
AABB	0.06720	1498
ABAB	-0.64938	1913
ABBA	-0.03617	1700
ABBB	0.00470	2127
BAAB	0.06172	1476
BABB	-0.00078	1618
BBBB	-0.13974	1949
AAAAA	-2.84828	3660
AAAAB	-1.58944	4197
AAABA	-2.44493	3715
AAABB	-0.54688	2511
AABAA	-2.53170	5068
AABAB	-1.34774	5362
AABBA	-0.92662	3189
AABBB	0.04017	3069
ABAAB	-1.37647	3359
ABABA	-2.22020	3719
ABABB	-0.61680	2797
ABBAB	-0.00565	3584
*ABBBA**	0.03870	2860
ABBBB	-0.06596	4187
BAAAB	-0.52108	4004
BAABB	0.09621	2220
BABAB	-0.64803	3798
BABBB	-0.18266	3504
BBABB	-0.24020	3356
BBBBB	-0.45266	4236

Table 2.5. Ground-states found by the inductive search. The predictions of our algorithm differ only for *ABBBA*.

CHAPTER 3

Evolutionary Constraints Handling for Problems with Explicitly Defined Constraints

3.1. Handling Additional Constraints in Combinatorial Optimisation Problems

Many real world problems can be cast as modifications of well known *NP*-complete problems where some additional constraints are explicitly defined to reflect the physical nature of the problem. In this section we deal with one such problem: the fault coverage test code generation problem, an instance of which is provided by Rolls Royce and Associates Ltd.

The lack of a uniform methodology for handling infeasible points [Michalewicz, 95] largely predetermines the current best practice – the investigation of some problem-specific operators which search within the feasibility boundary in an efficient way [Michalewicz, 96]. The idea is based on the seemingly reasonable assumption that in real world problems the constraints and the objective functions are conflicting and therefore, the constraint global solution lies on the boundary of the feasible region. We utilise this approach for the fault coverage test code generation problem and show how to derive the feasible region and design feasibility preserving operators that map the feasibility region onto itself.

3.1.1. The Test Pattern Generation Problem

We will be interested in generating tests for *combinational* circuits, which have no feedback loops or memory elements. Fig. 3.1. shows a simple combinational circuit. A *test pattern* for a potentially defective circuit is a set of inputs for the circuit that will cause the circuit outputs to be different if the circuit is defective than if it is defect-free. To derive the input set, we must have some model for possible defects (faults) in the circuit. One of the most popular models within the existing testing systems is the *single stuck-at* model. In this model, a defective circuit is assumed to behave as if it were defect-free, with the exception of one wire that is tied to either a logic 0 or a logic 1 (instead of correctly varying as a function of the circuit inputs). Logically equivalent inputs may fail independently. For example, in fig. 3.2. A_1 can be stuck-at 1 while A_2 takes on the value 0.

To generate a test pattern for a circuit with a wire stuck at 1, we must ensure that the wire in question would take on the logic value 0 in a correctly functioning circuit. If this is not the case, the circuit outputs will be the same whether or not the circuit is malfunctioning because the faulty and the good circuit would carry the same values. In fig. 3.1. line D is labelled with 0/1 to denote that line D is the site of a fault such that D will carry the value 0 if the circuit is functioning correctly and will carry the value 1 if the circuit is defected (i.e., *faulted*). When a line has a different value in the faulted and unfaulted circuits, it is said to have a *discrepancy*. Fig. 3.3. shows a test pattern that detects D stuck-at 1 and fig. 3.4. shows a test pattern that does not detect D stuck-at 1. We say that the test pattern of fig. 3.3. *covers* D stuck-at 1 and the test pattern of fig. 3.4. fails to cover D stuck-at 1. The test pattern generation problem is known to be *NP*-complete [Fuiwara and Toida, 82]. This fact can be easily demonstrated by showing that 3SAT (i.e., an instance of the satisfiability problem where each clause is allowed to have only three variables) [Cook, 71] is polynomial-time reducible to test pattern generation. First, we take a 3CNF (conjunctive

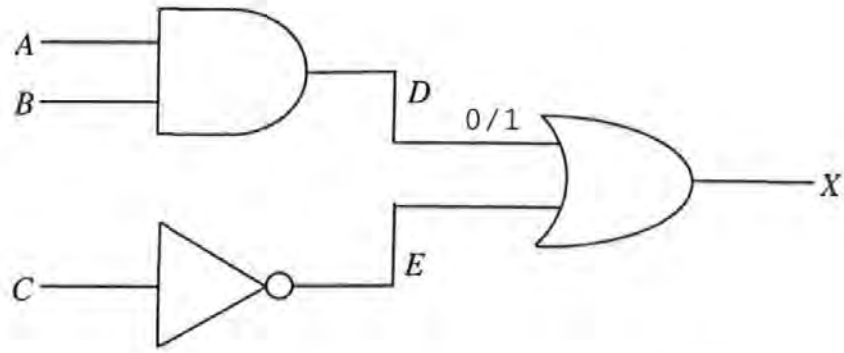


Fig. 3.1. Combinational circuit with D stuck at 1.

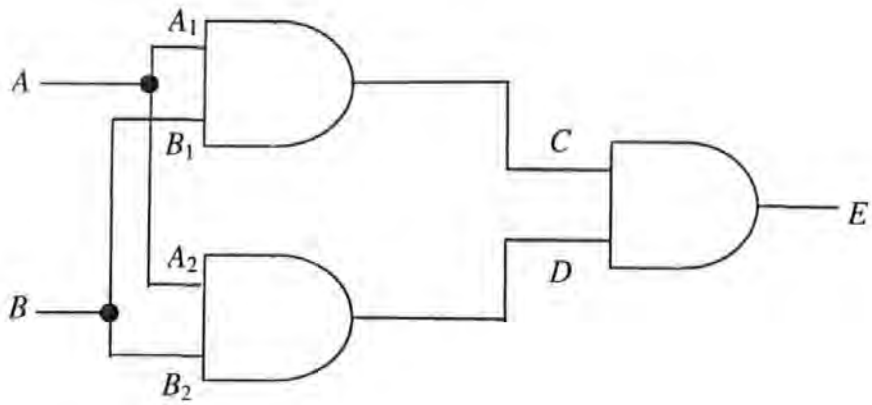


Fig. 3.2. Line A_1 can fail independently from line A_2.

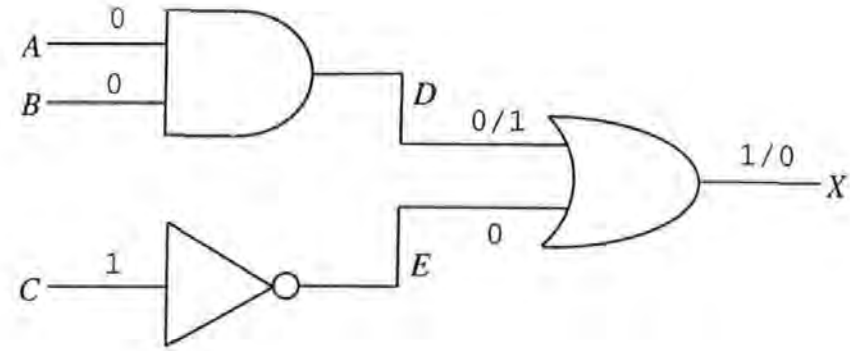


Fig. 3.3. Test pattern covering D stuck-at 1.

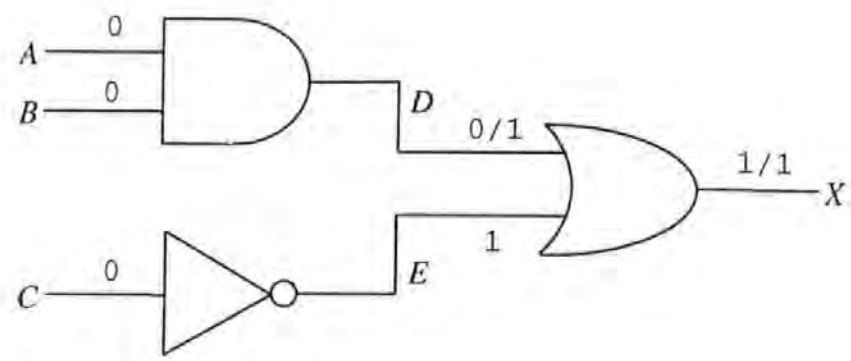


Fig. 3.4. Test pattern not covering D stuck-at 1.

normal form) formula (also known as a product of sums formula where each sum has at most three literals) and naively build the circuit corresponding to it. We can do this by creating one OR gate for each clause and feed the outputs of all the OR gates into one AND gate. In fig. 3.5. we show a circuit corresponding to the 3CNF formula $(A+\overline{B}+C)\cdot(A+B+\overline{C})$. Next, we generate a test pattern for the output of the circuit stuck at 0. If it were possible to generate the test pattern in polynomial time, it would be possible to satisfy a 3CNF formula in polynomial time.

It would be possible to generate a test pattern in linear time if it were not for *reconvergent fanout* [Ibarra and Sahni, 75]. In a combinational circuit, reconvergent fanout occurs whenever there is more that one path of logic elements between any two lines in the circuit. For example, in fig. 3.5., there is more than one path between line *A* and line *X*. The presence of reconvergent fanout introduces potentially unsatisfiable dependencies into the problem of test pattern generation.

The gate-level Automatic Test Pattern Generation (ATPG) problem has been approached in two major ways: algebraic techniques based on Boolean Differences, literal proposition, etc., and path sensitisation techniques that operate on the circuit topology like the D-algorithm [Roth, 66] and PODEM [Goel, 81]. In the first set of approaches, the circuit under test is typically represented by some form of a switching function like a truth table, sum-of-products expressions and Karnaugh maps. Test pattern generation is carried out by the manipulation of these representations. Until recently, algebraic techniques have not found favour because these techniques did not scale well with the size of the circuit under test. The main problems were those of generating the representations from circuit netlists and their manipulation when there were a large number of input variables and internal nodes. These problems have been alleviated to a great extent with the renewed interest in the use

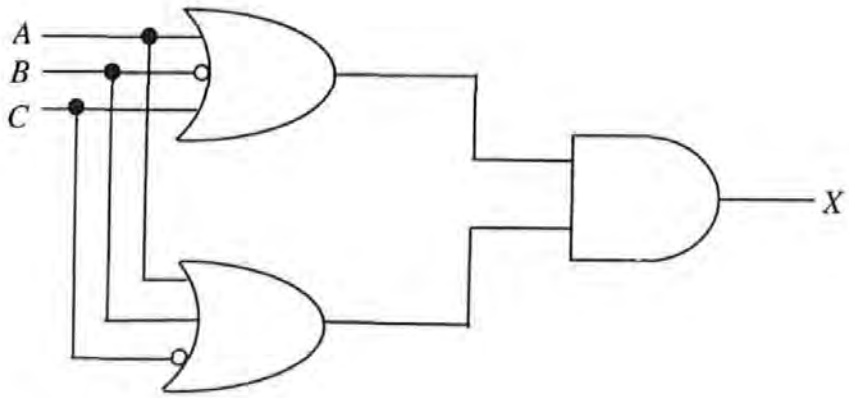


Fig. 3.5. Circuit corresponding to the formula $(A + \overline{B} + C) \cdot (A + B + \overline{C})$.

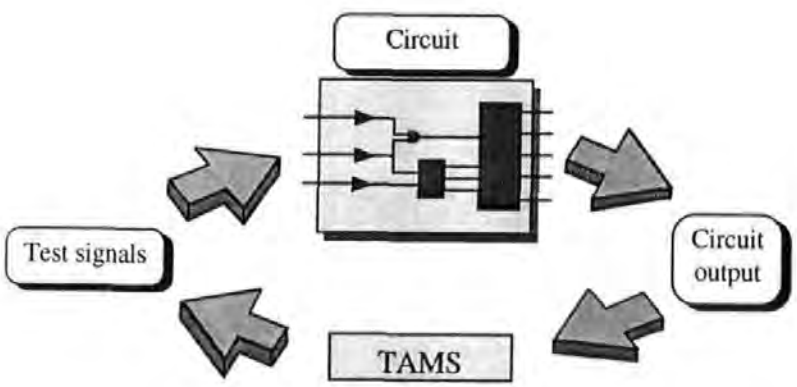


Fig. 3.6. Overview of test and monitoring system (TAMS)

of Ordered Binary Decision Diagrams (OBDDs) for the representation and manipulation of combinational logic. Some of the more significant work done in this area of ATPG includes CATAPULT [Gaede *et. al.*, 86], WAVE [Ross and Mercer, 90], and TSUNAMI [Stanion and Bhattacharya, 91].

Path sensitisation based ATPG algorithms operate on the circuit topology. ATPG is the problem of assigning values to inputs such that the circuit output(s) contain different values for the fault-free and faulty cases. The path sensitisation algorithms implicitly or explicitly search the entire space of input vectors to find a test. The current problems with this approach is that faults which are hard-to-detect for one algorithm may not be so difficult for another. This situation occurs because it is the specific decisions an algorithm makes and not the function the circuit implements that make it difficult for the algorithm to find a test for some fault. Popular algorithms that utilise the path sensitisation approach include D-algorithm [Roth, 66], PODEM [Goel, 81], FAN [Fujiwara and Shimono, 83], and SOCRATES [Schulz *et. al.*, 88].

3.1.2. The Test and Monitoring System and the Fault Coverage Code Generation Problem

After defining the test pattern generation problem we can now proceed with the definition of the fault coverage test code generation problem. Test codes (a set of input test vectors and a set of expected output vectors) are an integral part of the Test and Monitoring Systems (TAMS). TAMS are widely used for real-time testing of the functionality of electronic circuits (fig. 3.6). Basically they operate by regularly initiating a test cycle on the circuit and monitoring the fault status. For reliability reasons new circuits cannot be used until the fault coverage test code is updated with a new set of comprehensive test vectors (fig. 3.7).

The update is achieved with the help of a fault analysis process used to determine the fault detection coverage of a particular design. The fault analysis process for a design involves the optimisation of the input stimulus to fully exercise all components to increase the testability. Usually the fault analysis process fits within the product development cycle after the initial functional verification of the design and before the physical hardware testing of the product. However, due to other design considerations, here we face the problem of maximising the fault coverage for an already specified circuit. The amount of fault coverage within a design depends on the following two factors: (1) comprehensiveness of the test code, and (2) inherent testability of the logic design. In this section we concentrate on the first factor and formulate the problem of finding an effective set of input test vectors as a *search problem*:

> Given a set of patterns of the form `*1**10**0`, where `*` is a *don't care* symbol, find a set of N binary vectors that maximises the coverage of the given patterns. Coverage is defined in terms of Boolean matching.

3.1.3. Constraint Handling: Deriving the Generators of the Feasible Region and Designing Feasibility Preserving Operators

Usually there are various constraints imposed on the test codes. For example, the size of the test code may be constrained by hardware requirements of the test and monitoring system. There may also be a number of constraints concerning the possible combinations of input signals. The task is to automate the process of finding the most comprehensive test code, i.e., the code maximising the fault coverage (fig. 3.7).

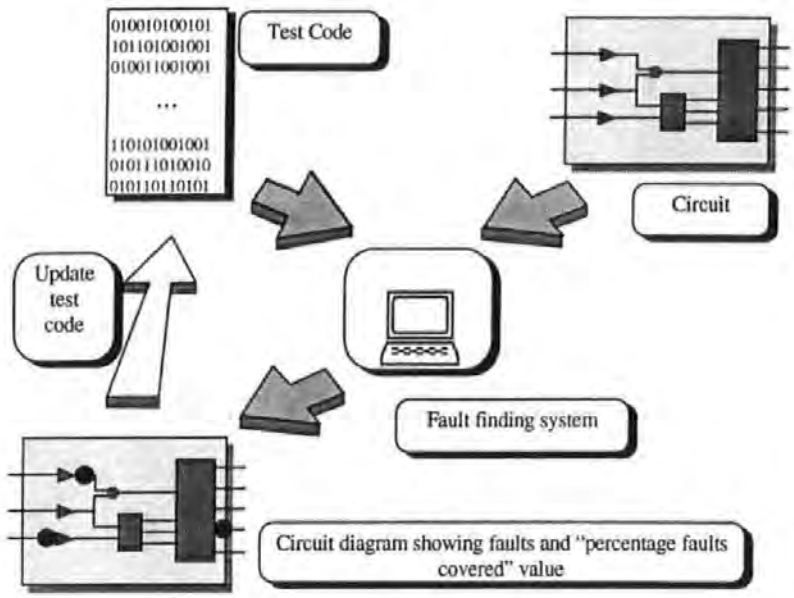


Fig. 3.7. The process of finding the most efficient fault coverage test code: The circuit is modelled and faults are simulated. Using information from the fault analysis the task is to design the most comprehensive test vectors (the white arrow).

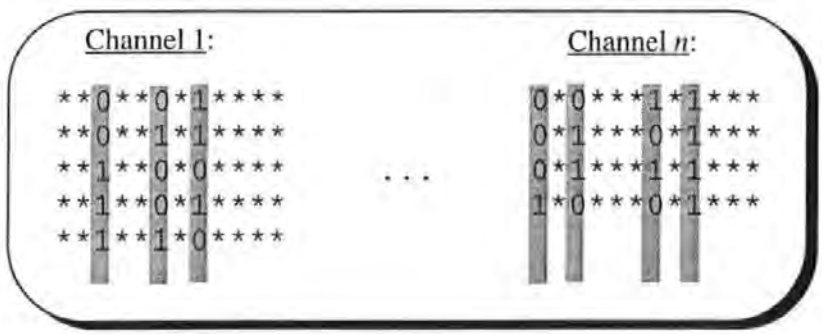


Fig. 3.8. A set of legal test codes is defined by the legal states of a number of logical channels.

The requirement that the number of test vectors must be exactly N is represented directly by the coding scheme of the problem. A sample from the associated fitness landscape of the search problem would consist of N vectors each of length m bits.

The second type of constraints impose strict requirements on the possible combinations of values within each individual test vector. The designers of the circuit define the set of legal combinations in terms of the legal states of a number of channels (fig. 3.8). Each channel is a logical grouping of input bits (for example, bits No. 2, 5, and 7 could form logical channel 1). Collectively the legal states of all channels define a set of legal (*supporting*) templates of the form:

```
1*0**1011***
```

where * is a don't care symbol. Each template could be viewed as a generator of a particular fraction (subspace) of the original search space. Therefore, the set of all legal templates defines the feasible region. The existence of a such closed form description of the feasible region greatly influences the selection of a constraint handling technique. In our case, it seems appropriate to maintain a population of legal samples by designing feasibility preserving search operators.

When applied to a feasible point(s), a feasibility preserving operator always produces another feasible point(s). For the test code generation problem we have designed two versions of mutation and one of crossover which comply with the selected constraint handling technique.

mutation 1: (*i*) find the supporting template of the parent chromosome, and (*ii*) apply uniform mutation to the values of the don't care bits.

```
100101011011    parent chromosome
1*0**1011***    parent's supporting template
110111011001    offspring
```

mutation 2: (*i*) find the supporting template of the parent chromosome, (*ii*) change it by randomly selecting another supporting template while keeping the values of the don't care symbols.

```
100101011011    parent chromosome
1*0**1011***    parent's supporting template
0*1**0011***    new supporting template
001100011011    offspring
```

crossover: (*i*) find the supporting template of both parents and (*ii*) apply uniform crossover to the don't care bits.

parents	100101011011	011010011001
templates	1*0**1011***	0*1**0011***
offspring	110111011001	001000011011

3.1.4. Utilisation of the Inductive Search Approach

In general the inductive approach generates a solution step by step, beginning from the so called *base* of the induction and at each step following an *induction rule* to update (i.e., induce) the solution. In mathematics induction is a rigorous proof technique while in the context of adaptive search it is used to approximately induce a solution to a particular problem. Previous research [Bilchev and Parmee, 96d][Bilchev and Parmee, 96e] well justifies the potential power of the inductive approach in the context of search.

Applying the inductive approach to the fault coverage code generation problem requires a slight reformulation of the problem. The original problem is:

> Given a number *N* (the maximum number of fault coverage test vectors) find a sequence of *N* test vectors that maximizes the fault coverage.

It can be easily reformulated as:

For each k = 1 to N find a sequence of k test vectors that maximizes the fault coverage.

While being the same problem the inductive formulation also gives meaning to intermediate solutions. Suppose for example that for some k, $1<k<N$, we know a sequence of test vectors which gives *satisfactory* fault coverage. The term satisfactory means that with k test vectors we couldn't expect to cover many more faults than those already discovered by the sequence. This is a relative judgement regarding the particular circuit and can serve as an efficient stopping condition instead of the usual maximum number of generations (ref. to step 6 of the algorithm). Now if for k test vectors we have already achieved a satisfactory level of fault coverage, the task is to find a satisfactory fault coverage level for k+1 test vectors. The main power in the inductive approach is the assumption that the satisfactory fault coverage level for k+1 test vectors could be derived from the satisfactory level of the k test vectors. If this assumption is true then it produces an efficient search engine with computational complexity determined only by the computational complexity of the inductive step.

The Inductive Genetic Algorithm (IGA) combines the evolutionary search engine with an inductive fitness function. The overall structure of the IGA is as follows:

1. Initialize a partial solution for N = 1 (i.e., a sequence of one test vector only)

   100101011011

2. For k = 2 to N do (search for the best k*th* vector that complements the already existing partial solution)

3. Initialize a population of test vectors

   011001001110 ... 010010111010

4. Add each test vector to the partial solution, evaluate it and assign fitness

```
100101011011                    100101011011
011001001110          ...       010010111010
```

5. Reproduce according to the fitness obtained in 4. Typical operators include versions of mutation and crossover. Update the population.

6. If not end of generations, goto 4

7. Update the partial solution, increment k, goto 2

Steps 3, 4, 5, and 6 constitute a genetic algorithm. Steps 1, 2 and 7 implement the inductive approach. The overall algorithm could also be viewed as a genetic algorithm with dynamic fitness function, i.e., the fitness function changes at each generation.

3.1.5. Experiments

In order to develop and tune our search strategy we have designed an efficient simple model of the fault population of a *virtual circuit*. As the search engine should be generic enough to run on a variety of circuits such a circuit abstraction is well justified. In our model each fault population consists of a number of faults with associated fault identifying patterns (we can assume that the identifying patterns are produced by an ATPG system as described in section 3.1.1. for a particular circuit). For the experiments, faults are not allowed to have illegal identifying patterns (otherwise they are "intrinsically" untestable). A simple fault population with five faults may look like:

Fault No.	Identifying Pattern
1.	*01*********
2.	*01*0*******
3.	101*******10
4.	0*0*11011***
5.	**1*1101****

where "*" is a *don't* care symbol. The task of the fault coverage problem is to find a "generalist" population of test vectors covering as many faults as possible. For example, the following two test vectors cover 80 percent of the fault population:

Test Vector No.	Identifying Pattern	Fault Covered
1.	101101001110	No. 1,2,3
2.	000111011001	No. 4

The associated fitness function with our model is

$$f = \frac{n}{N} \cdot 100\% \tag{3.1}$$

where n is the number of covered faults and N is the number of all faults in the fault population.

Fig. 3.9. shows the performance of the Inductive Genetic Algorithm (IGA) on a fault coverage problem consisting of 200 possible faults (Appendix E). The number of input test vectors is 24. The search effort at each inductive step controls the trade-off between the computational complexity and the expected quality of results. The family of all possible trade-off points define the *performance trade-off* front. It is a measure of the expected gain of the quality of results as a function of the computational expense.

Fig. 3.10. compares the IGA approach with a simple genetic algorithm (i.e., the induction step is turned off and each chromosome has fixed length of 24 vectors each 12 bits long) applied to the whole sequence of test vectors. The same test problem as in the previous experiment is used. From fig. 3.10. it can be clearly seen that the IGA outperforms the simple genetic algorithm. One possible explanation is that the IGA considerably reduces

the search space by dividing it into disjoint inductive search subspaces, whereas the simple GA works on the huge original search space (approx. $2^{24\times12}$).

Fig. 3.11. shows the performance of the IGA as a function of the number of input test vectors. The same fault population of 200 faults as in the previous experiments is used. The IGA is able to find a set of 67 input test vectors that cover 100% of the fault population. If the hardware requirements allow the TAMS to use 67 test vectors then 100% fault coverage could be achieved. However, in real world problems there are hard constraints imposed on the design task. For example, in our particular TAMS test code generation problem the number of input test vectors must be 24. Therefore, our objective is to design maximally comprehensive set of 24 test vectors (the design of the logic of the circuit is already fixed, so we regard the test coverage code generation problem as a search problem and do not address the inherent testability properties of the logic design).

Fig. 3.12. shows the effect of the constraints on the performance of the search engine. There are two graphs, each corresponding to a particular set of constraints. Each set of constraints is determined by a table of legal states. Legal table 2 is derived from legal table 1 by reducing the number of legal states in channel 1. Therefore, legal table 2 corresponds to a more constrained instance of the fault coverage test code generation problem. Imposing constraints on the problem is somewhat equivalent to introducing *inherent untestability* in the circuit design. Certain legal identifying fault patterns can no longer be allowed and therefore, the corresponding faults they cover cannot be detected. As can be seen from fig. 3.12 this fact significantly influences the fault coverage.

In this section we used a real world problem to demonstrate a very efficient and well known constraint handling technique. It consists of defining the feasible region in terms of

the independent variables and designing feasibility preserving operators (i.e., operators that map the feasible region onto itself). The existence of such a closed form description of the feasible space leads to a minimal redundancy problem representation [Radcliffe, 91] and could significantly reduce the search space. Currently the feasibility preserving constraint handling technique is being applied successfully to the optimisation of real valued functions and linear constraints [Michalewicz, 92], and for combinatorial problems (chapter 2). In this section we have shown that the approach is quite generic and applied it to the fault coverage test code generation problem [Bilchev and Parmee, 96] with additional explicitly defined constraint requirements imposed by the designers of the circuit's logic.

We consider the reduction of a search space to be one of the most efficient approaches for solving any search problem. This idea has been fundamental for many of the existing search methodologies, including branch-and-bound, clustering, etc. In this section we also proposed to integrate the search space reduction approach (implemented as an inductively defined fitness function) with an evolutionary search engine. The idea has already produced successful results when applied to optimisation of real valued continuous functions (chapter 2).

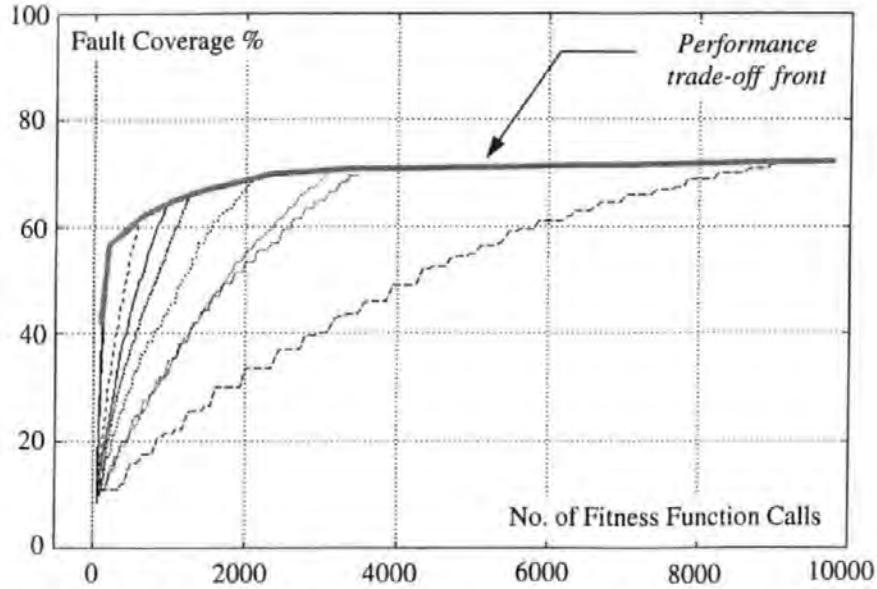


Fig. 3.9. Runs of the Inductive Genetic Algorithm for eight different control parameter settings (number of generations per inductive step). The family of all possible control parameter settings define the *performance trade-off front*, which is a measure of the trade-off between computational complexity and quality of results. The number of input test vectors is fixed to 24.

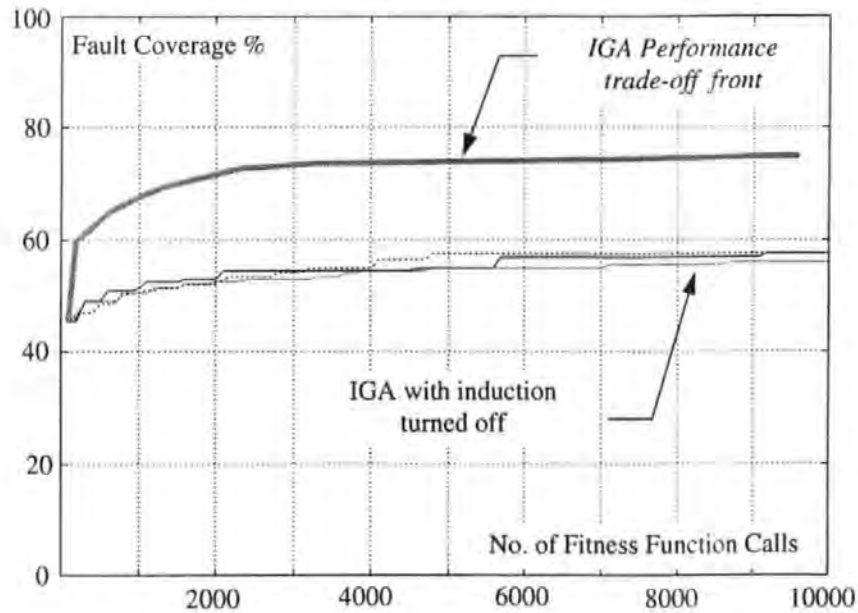


Fig. 3.10. Comparison between the Inductive Genetic Algorithm (IGA) and the simple genetic algorithm (i.e., no induction).

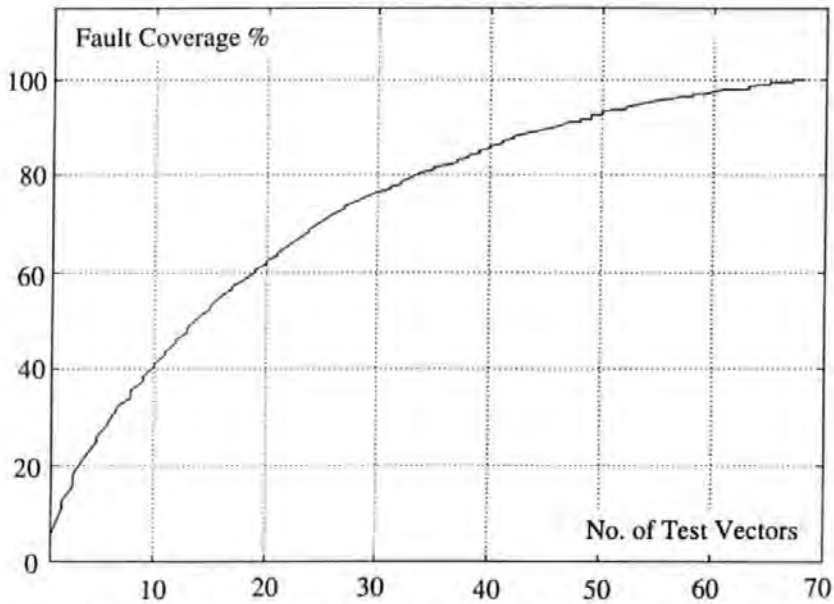


Fig. 3.11. The fault coverage as a function of the number of input test vectors. There are 200 faults in the fault population. An Inductive Genetic Algorithm is used to find the best set of test vectors. The number of fitness function calls in order to cover 100 percent of the fault population is approximately 10,000 and required 67 input test vectors.

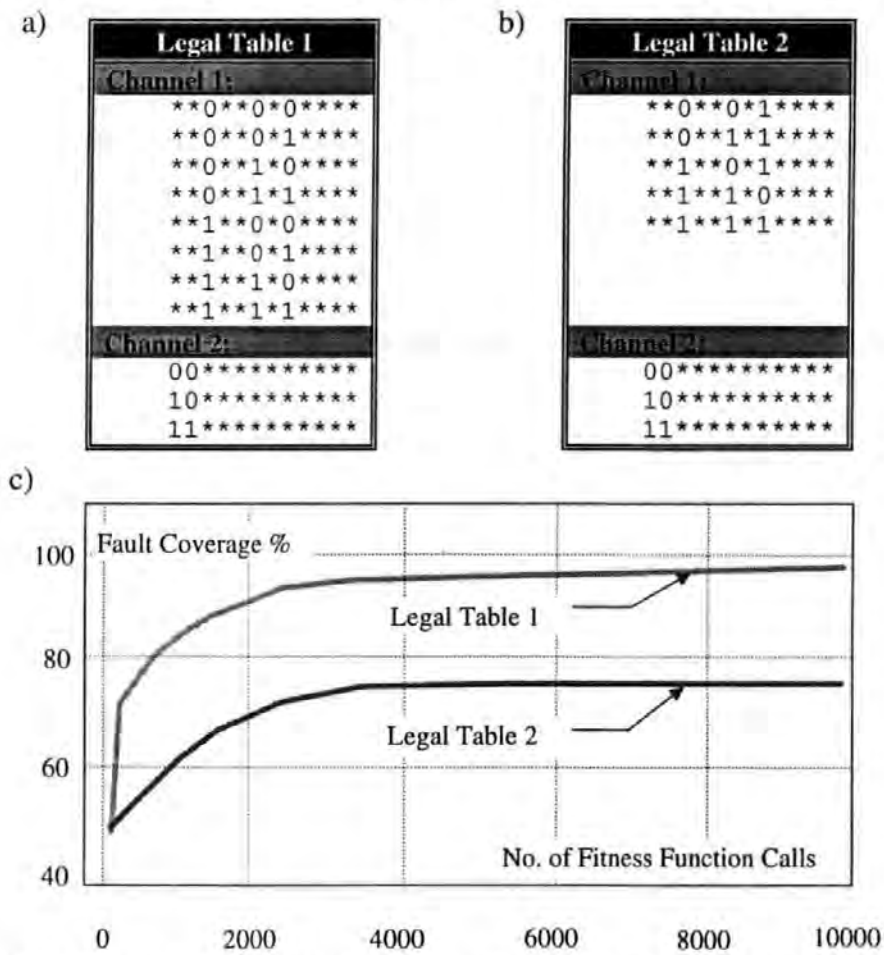


Fig. 3.12. Effects of constraints on the performance trade-off front of the IGA. The effect of imposing constraints on the fault coverage problem (Legal Table 2) is equivalent to introducing inherent untestability in the circuit.

CHAPTER 4

Evolutionary Constraint Handling for Problems with Implicitly Defined Constraints

4.1. The Ant Colony Search Model for Functions of Continuous Variables

In this chapter we extend the ant colony search model introduced in chapter 2 to deal with constrained optimisation of functions of continuous variables and present a number of empirical results.

It is a well established belief that if no *a priori* knowledge about the problem at hand f is incorporated into the search algorithm, the problem scales exponentially with its dimension [Kowalik, 68]. This phenomenon, known in optimisation as the "curse of dimensionality" [Fletcher, 87], led to the abandonment of direct search methods in favour of those using some *a priori* knowledge (assumptions) about f.

In this section we extend the ant colony search model introduced in chapter 2 to deal with continuous domains. The assumptions that we will make are that (1) new samples of f should most often be obtained in the vicinity of previous, high-performance samples, (2) the number of new samples in the vicinity of a previous sample must depend on the observed value of f at that sample, and (3) the breadth of the sampling distribution around

the previous samplings should decrease as the global optimum is approached. The applicability of the ant colony search model to engineering design problems is our major concern. This typically involves applications to highly-dimensional and multi-modal problems with various kinds of constraints that are imposed in order to satisfy *a priori* defined performance criteria or behaviour.

The ant cycle algorithm (chapter 2) is not appropriate for continuous space searches. Keeping analogy with the foraging strategies of ant colonies we suggest an ant colony model applicable to continuous spaces. The main difficulty is how to model a continuous neighbourhood with a discrete structure. The strategy we have adopted is to represent a finite number of directions as vectors starting from a base point, called *the nest.* As potentially all of the continuous search space has to be covered, these vectors are evolving in time according the ants' fitness (fig.4.1.).

The structure of the ant colony algorithm is shown in fig. 4.2. Before the algorithm begins we have to initialise the nest structure by generating uniformly random starting directions as shown in fig. 4.3. Next we define a search radius R, which determines the maximum extent of the subspace to be considered in each generation (cycle). Then `initialize A(t)` sends ants in various directions at a radius not greater than R (see fig. 4.3); `evaluate A(t)` is a call to the objective function for all ants; `add_trail A(t)` is proportionally (to the ants' fitness) adding trail quantity to the particular directions the ants have selected, `send_ants A(t)` sends ants by selecting directions using a roulette wheel selection on the trail quantity and making a random step from the location of the best previous ant that had selected the same direction, `evaporate A(t)` decrements the trail. The random step is implemented as:

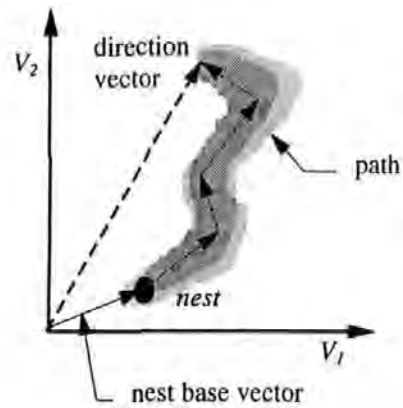


Fig. 4.1. A vector representing the direction towards the end of the actual path between the nest and the "food source" after four steps.

```
procedure Ant Colony Algorithm
begin
     t ← 0
     initialize A(t)
     evaluate A(t)
     while (not end_cond) do
     begin
          t ← t + 1
          add_trail
          send_ants
          evaluate
          evaporate
     end
end
```

Fig. 4.2. The structure of the Ant Colony Algorithm. `A(t)` is a data structure representing the nest and its neighbourhood.

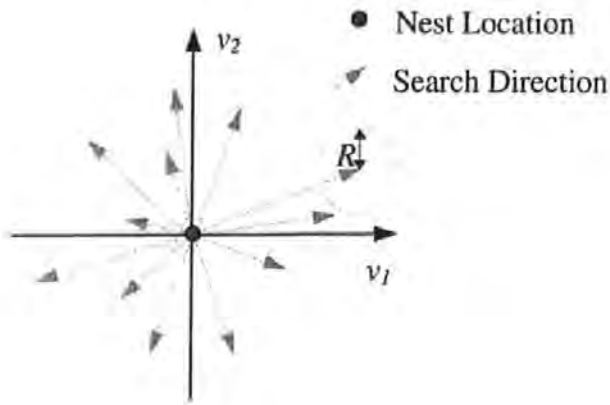


Fig. 4.3. Two dimensional Nest neighbourhood model with twelve search directions. Each direction evolves in time according to the fitness of the ants that have selected it.

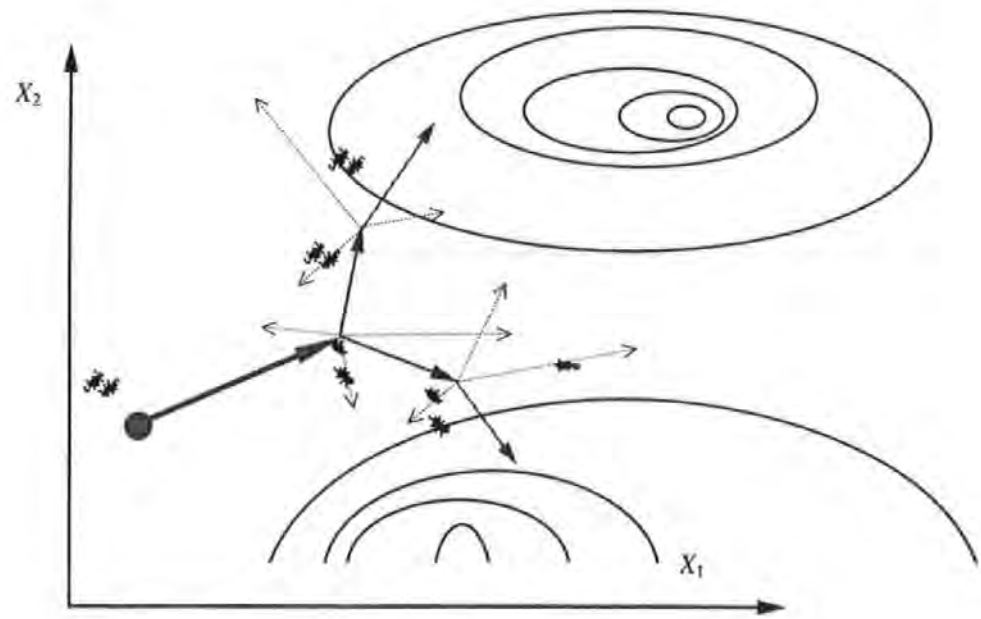


Fig. 4.4. Various directions are represented in a two dimensional search space.The bold lines show directions with high trail value. The dashed lines show unsuccessful sampling (i.e., samples that result in lower cost function value).

$$\Delta(t,R) = R \cdot (1 - r^{[1-t/T]^b}) \qquad (4.1)$$

where R is the search radius, r is a random number from [0..1], T is the maximal generation number, and b is a system parameter determining the degree of non-uniformity. $\Delta(t,R)$ returns a value in the range [0..R] such that the probability of $\Delta(t,R)$ being close to 0 increases as t increases. R is determined by the extent of the search subspace we want to cover during the run and corresponds to scaling of the ant colony model. During the run if certain directions do not result in improvement, they do not participate in the trail adding process and the reverse (evaporation) process diverts attention away from them.

The proposed ant colony model comprises three levels of abstraction. The lowest level is that of the *individual search agent*. It describes the employed individual search strategy, e.g., stochastic hill climbing, steepest descent, line search, etc. The middle level defines *co-operation* among agents which generally consists of a joint search effort in a certain direction. The highest level is the *meta* level which defines some high order *a priori* assumptions about the nature of the fitness landscape. At this stage it is important to notice the difference between direction and path in our model. The *direction* simply implies a physical location like north-east, etc. Due to self-organisation certain directions turn into *paths* as more and more trail is accumulated onto them (i.e., more and more ants are attracted). When no further improvement can be made along a particular path, no more trail is laid onto the path and the evaporation process turns it back to a direction. Stated in other words a path is a direction with high trail value.

Individual Search Level

The current utilised individual search strategy is purposely kept simple enough in order to reveal the power of co-operation. It consists of stochastically selecting a search direction

and making a step with size calculated by $\Delta(t,R)$. During the next generation (cycle) only the best new samples will be considered in the trail adding process. The overall individual search strategy can be viewed as stochastic hill-climbing.

Co-operation Level

The ants select randomly a direction to search with probability:

$$P_i(t) = \frac{[\tau_i(t)]}{\sum_k [\tau_k(t)]} \tag{4.2}$$

where τ_i is the trail on direction i. If the ants return with a higher fitness value they add trail on the selected direction. The added trail is proportional to the fitness value and it changes $P_i(t)$. Thus some directions become more attractive than others. A highly attractive direction eventually turns into a path as more and more agents follow it (fig. 4.4.). All of the agents from a particular path contribute to the joint search effort on that particular direction.

An evolution of the ant colony dynamics in time is shown in fig. 4.5. The experiment uses one hundred ants, b=2 and R=0.1. The selected fitness function is:

$$\mathrm{F2}(x) = e^{-2(\ln 2)\left(\frac{x-.1}{.8}\right)^2} \sin^6(5\pi x) \tag{4.3}$$

The number of ants attracted to each of the peaks at each generation depends on the peak's fitness. Better peaks (maximisation) attract more ants at the beginning of the evolution and less ants at the end. The dynamics of the ant colony search model are sensitive to the

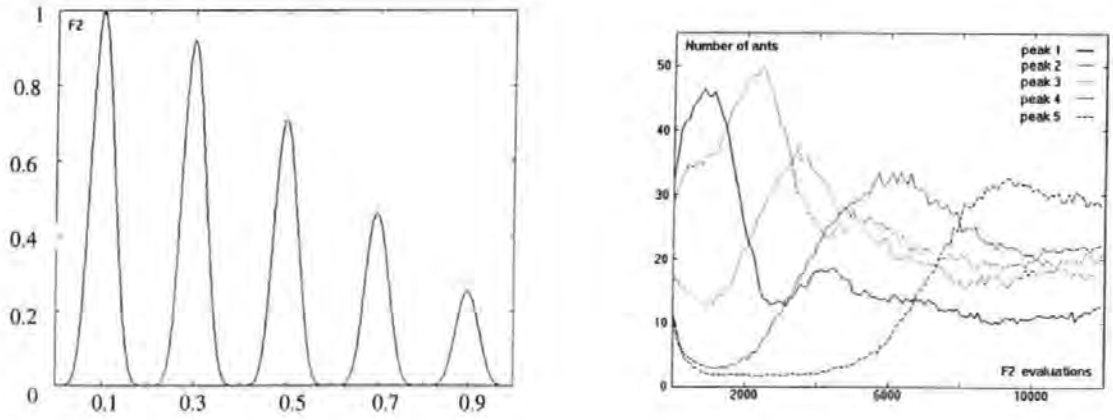


Fig. 4.5. Dynamics of the ant colony search model. The first figure shows the test function and the second reveals the time response of the ant colony search model.

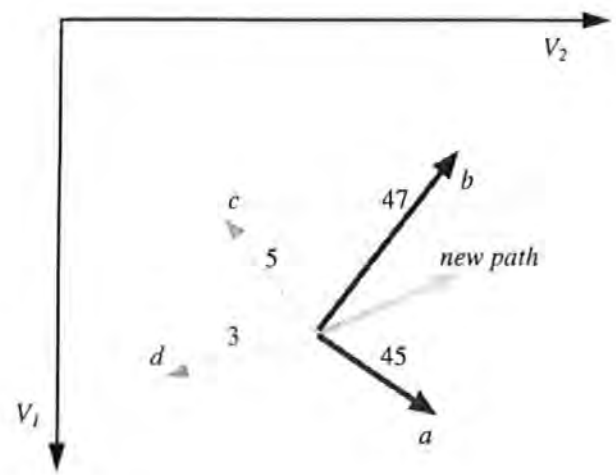


Fig. 4.6. A new path determined by trail diffusion from paths *a* and *b*.

evaporation parameter (i.e., how quickly the trail added by other ants is evaporated). This makes the ant colony a promising metaphor for fitness functions, fostering both co-operation and selfishness; however, this can make the tuning of the parameters more difficult.

Meta Level

The meta level can be defined as the mutual interaction between paths or some other kind of heuristics. For example, the intersection of trail diffused from two paths can be considered to form a new path which attracts agents. In this respect the effect of trail diffusion in the ant colony model is analogous to arithmetic crossover in GAs (fig. 4.6.). At present the meta level is not well understood and efficient meta rules are difficult to define. Current practice includes functions with relatively little variable interaction where exchanging variables from several partial solutions could potentially pay off. An example of how to define non-trivial meta rules using problem specific knowledge is shown at the end of section 4.3.

4.2. Handling Constraints in the Ant Colony Search Model

The classical treatment of constrained optimisation defines the problem as:

$$\begin{aligned} &\min_{X} F(X) \\ &s.t. \quad g_j(X) \leq 0 \end{aligned} \tag{4.4}$$

where $j=1,...,N$. The constraint functions g_j are assumed to be defined explicitly in terms of the (design) variables X. The utilisation of the ant colony model for constrained optimisation is mainly concerned with the representation of the constraints. We propose a very simple model: the constraint violation determines the acceptability of a point from the

search space, i.e., a point with high constraint violation is not accepted as a "food source" regardless of the value of the objective function. As the ant colony evolves the constraints are tightened and previously acceptable food sources vanish from the view of the ant agents. The acceptable constraint violation is implemented as a linear function T depending on (an estimate of) the ratio of the feasible region to the overall search space. The smaller the ratio, the greater the slope. To summarise, the modifications made to the ant colony algorithm (fig. 4.2.) consist of a new rule for laying trail: if a point X is feasible or violates the constraints less than a threshold T then add trail according to the value of the cost function, otherwise add trail according to the constraint violation (smaller constraint violation is attributed higher fitness); always attribute higher trail to feasible points by ranking.

4.3. Experiments and Comparisons with Existing Evolutionary Constraint Handling Techniques

The ant colony search model for constrained optimisation is tested on the following test cases proposed by in [Michalewicz, 95]:

test case #1: $F(X) = 5x_1 + 5x_2 + 5x_3 + 5x_4 - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i$

subject to:

$$
\begin{aligned}
&2x_1 + 2x_2 + x_{10} + x_{11} \le 10,\\
&2x_1 + 2x_3 + x_{10} + x_{12} \le 10,\\
&2x_2 + 2x_3 + x_{11} + x_{12} \le 10,\\
&-8x_1 + x_{10} \le 0,\\
&-8x_2 + x_{11} \le 0,\\
&-8x_3 + x_{12} \le 0,\\
&-2x_6 - x_7 + x_{11} \le 0,\\
&-2x_4 - x_5 + x_{10} \le 0,\\
&0 \le x_i \le 1, i = 1,\ldots,9,13,\\
&0 \le x_i \le 100, i = 10,11,12.
\end{aligned}
$$

The problem has 9 linear constraints; the cost function is quadratic with global minimum at X=(1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 1), where $F(X)$=-15.

test case #2: $F(X) = x_1 + x_2 + x_3$

subject to:

$$1 - 0.0025(x_4 + x_6) \geq 0,$$
$$1 - 0.0025(x_5 + x_7 - x_4) \geq 0,$$
$$1 - 0.01(x_8 - x_5) \geq 0,$$
$$x_1 x_6 - 833.33252 x_4 - 100 x_1 + 83333.333 \geq 0,$$
$$x_2 x_7 - 1250000 - x_3 x_5 + 2500 x_5 \geq 0,$$
$$100 \leq x_1 \leq 10000,$$
$$1000 \leq x_i \leq 10000, i = 2,3,$$
$$10 \leq x_i \leq 1000, i = 4,\ldots,8.$$

The problem has 3 linear and 3 non-linear constraints; the cost function is linear and has its global minimum at X=(579.3167, 1359.943, 5110.071, 182.0174, 295.5985, 217.9799, 286.4162, 395.5979) where $F(X)$=7049.330923.

test case #3:

$$F(X) = (x_1 - 10)^2 + 5(x_2 - 12)^2 + x_3^4 + 3(x_4 - 11)^2 + 10x_5^6 + 7x_6^2 + x_7^4 - 4x_6 x_7 - 10x_6 - 8x_7$$

subject to:

$$127 - 2x_1^2 - 3x_2^4 - x_3 - 4x_4^2 - 5x_5 \geq 0,$$
$$282 - 7x_1 - 3x_2 - 10x_3^2 - x_4 + x_5 \geq 0,$$
$$192 - 23x_1 - x_2^2 - 6x_6^2 + 8x_7 \geq 0,$$
$$-4x_1^2 - x_2^2 + 3x_1 x_2 - 2x_3^2 - 5x_6 + 11x_7 \geq 0,$$
$$-10.0 \leq x_i \leq 10.0, i = 1,\ldots,7.$$

The problem has 4 non-linear constraints; the cost function is non-linear and has its global minimum at X=(2.330499, 1.951372, -0.4775414, 4.365726, -0.6244870, 1.038131, 1.594227) where $F(X)$=680.6300573.

test case #4: $F(X) = e^{x_1 x_2 x_3 x_4 x_5}$

subject to:

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 = 10,$$
$$x_2 x_3 - 5x_4 x_5 = 0,$$
$$x_1^3 + x_2^3 = -1,$$
$$-2.3 \le x_i \le 2.3, i = 1,2,$$
$$-3.2 \le x_i \le 3.2, i = 3,4,5.$$

The problem has 3 non-linear constraints; the cost function has its global minimum at X = (-1.717143, 1.595709, 1.827247, -0.7636413, -0.7636450) where $F(X)$=0.0539498478.

test case #5:

$$F(X) = x_1^2 + x_2^2 + x_1 x_2 - 14x_1 - 16x_2 + (x_3 - 10)^2 + 4(x_4 - 5)^2 + (x_5 - 3)^2$$
$$+2(x_6 - 1)^2 + 5x_7^2 + 7(x_8 - 11)^2 + 2(x_9 - 10)^2 + (x_{10} - 7)^2 + 45$$

subject to:

$$105 - 4x_1 - 5x_2 + 3x_7 - 9x_8 \ge 0,$$
$$-10x_1 + 8x_2 + 17x_7 - 2x_8 \ge 0,$$
$$8x_1 - 2x_2 - 5x_9 + 2x_{10} + 12 \ge 0,$$
$$-3(x_1 - 2)^2 - 4(x_2 - 3)^2 - 2x_3^2 + 7x_4 + 120 \ge 0,$$
$$-5x_1^2 - 8x_2 - (x_3 - 6)^2 + 2x_4 + 40 \ge 0,$$
$$-x_1^2 - 2(x_2 - 2)^2 + 2x_1 x_2 - 14x_5 + 6x_6 \ge 0,$$
$$-0.5(x_1 - 8)^2 - 2(x_2 - 4)^2 - 3x_5^2 + x_6 + 30 \ge 0,$$
$$3x_1 - 6x_2 - 12(x_9 - 8)^2 + 7x_{10} \ge 0,$$
$$-10.0 \le x_i \le 10.0, i = 1,\ldots,10.$$

The problem has 3 linear and 5 non-linear constraints; the cost function has its minimum at X=(2.171996, 2.363883, 8.773926, 5.095984, 0.9906548, 1.430574, 1.321644, 9.828726, 8.280092, 8,375927) where $F(X)$=24.3062091.

The ant colony search model gives better minima at the same level of constraint violation as compared to other existing state-of-the-art evolutionary constrained handling techniques [Michalewicz, 95 a]. Results are summarised in Table 4.1. For example, the best results described in [Michalewicz, 95 a] are -15.00, 8206.15, 681.11, 0.064 and 26.9 for test cases from 1 to 5 respectively.

Compared to other evolutionary methods for constrained optimisation the ant colony model shows excellent performance and quality of solution especially for the problems with non-linear constraints. A remarkable feature of the ant colony model is that the standard deviation of the solutions (averaged over 10 independent runs) is considerably less that the standard deviation produced by other evolutionary methods for constrained optimisation [Michalewicz, 95a].

The performance of the ant colony search model can be significantly increased if the individual search strategy is appropriately selected. For example, if the ant colony utilises a Sequential Quadratic Programming method [Lawrence *et. al.*, 96], instead of the simple stochastic hill-climbing, the performance on the five test cases could be dramatically increased. Results are shown in table 4.2. In this case the performance is so significantly increased because the test cases prove to be easy for the individual search strategy alone and no co-operation is necessary (actually, here co-operation slows down the program execution on a sequential machine as many of the agents will reach the same solution in parallel).

runs	test 1	test 2	test 3	test 4	test 5
opt.	-15.00	7049	680.6	0.054	24.30
1.	-14.39	7293	680.8	0.053	26.53
2.	-14.46	7624	680.8	0.054	25.83
3.	-14.76	7486	680.9	0.055	25.81
4.	-14.45	7674	680.8	0.066	26.04
5.	-13.93	8009	680.8	0.056	26.12
6.	-14.59	7343	681	0.055	25.78
7.	-14.46	7774	681	0.057	26.78
8.	-14.53	7650	680.8	0.055	25.72
9.	-14.33	8082	680.9	0.054	26.29
10.	-14.64	7704	681	0.056	25.72
aver.	-14.45	7663	680.9	0.056	26.06

Table 4.1. Results of running the Ant Colony model on the five test cases proposed in [Michalewicz, 95]. The assumed constraint violation accuracy is 0.01 for each constraint. The maximum number of fitness function calls is 50,000. Each fitness function call includes one call to the cost function and one call to the constraint function.

	no. of cost function calls	no. of constraint function calls	found value
test 1	50	0(*)	-15
test 2	58	381	7049.25
test 3	165	483	680.63
test 4	1186	5978	0.05394
test 5	37	383	24.306

Table 4.2. Results from applying an Ant Colony model utilising sequential quadratic programming as individual search strategy on the five test cases proposed by Michalewicz [Michalewicz, 95]. (*) Zero number of constraint function calls is possible in this case because the constraints are linear and they are eliminated analytically before the optimization begins.

Unfortunately, the selection of individual search strategy cannot always guarantee such an increased performance. It is potentially possible that much CPU time is allocated to the individual search (i.e., local exploitation) and little remains for the global exploration thus resulting in decreased overall performance. Such a scenario is quite plausible for highly multi-modal fitness landscapes. In this case other approaches, such as the definition of appropriate meta-control rules, seem very promising. An example of how to define effective meta-control rules is presented in the following paragraphs using the bump problem proposed by Professor A. Keane [Keane, 94]. The bumpy equation simulates a multi-peak optimisation problem. The objective function is defined as follows:

$$f(\bar{x}) = \left\{ abs(\sum_{i=1}^{m} \cos^4(x_i) - 2\prod_{i=1}^{m} \cos^2(x_i)) \right\} / \left\{ \sqrt{\sum_{i=1}^{m} i x_i^2} \right\} \qquad (4.5)$$

where the x_i, $i= 1,...,m$ are the variables (expressed in radians) and m is the number of dimensions. This function produces a series of peaks that get smaller with distance from the origin and which are nearly symmetrical about $x_i=x_j$, $i,j=1,...,m$. The optimisation problem is then defined as find x_i for $0<x_i<10$, $i=1,...,m$ to maximise the function subject to $\prod_{i=1}^{m} x_i > 0.75$ and $\sum_{i=1}^{m} x_i < 300$. A contour plot of the fitness function for the 2D bump problem is shown on fig. 4.7.

There are some interesting features about the bump problem which can be used as a basis for the definition of efficient meta-control rules. For example, the observation that the fitness function can always be improved by sorting the co-ordinates of a search point in decreasing order is implemented as a heuristic in the ant colony search model (i.e., at each generation the heuristic is applied to the best solution). It greatly increases the performance

of the search algorithm and empirically proves the fact that algorithms that use as much as possible relevant information about the problem at hand achieve better results (fig. 4.8.).

This section presented the ant colony metaphor as a high level description language for distributed searches. Current research identifies three levels of abstraction: the individual, the group, and the environment (landscape). Although easy to describe, such models are often mathematically intractable to analyse due to the non-linear coupling between the three levels.

We also empirically showed that incorporating prior knowledge into the search process significantly improves the performance. The ant colony metaphor proves quite useful when applied to engineering design problems as it enables the engineer to fully take advantage of the adaptive search paradigm and to easily implement constrained search as will be seen in chapter 6.

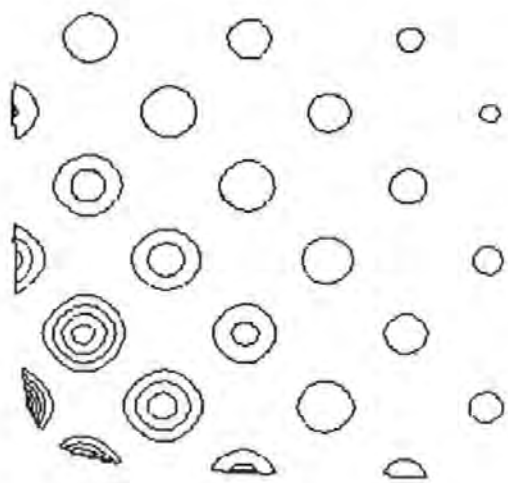


Fig. 4.7. A contour plot of the fitness landscape of a 2D bump problem.

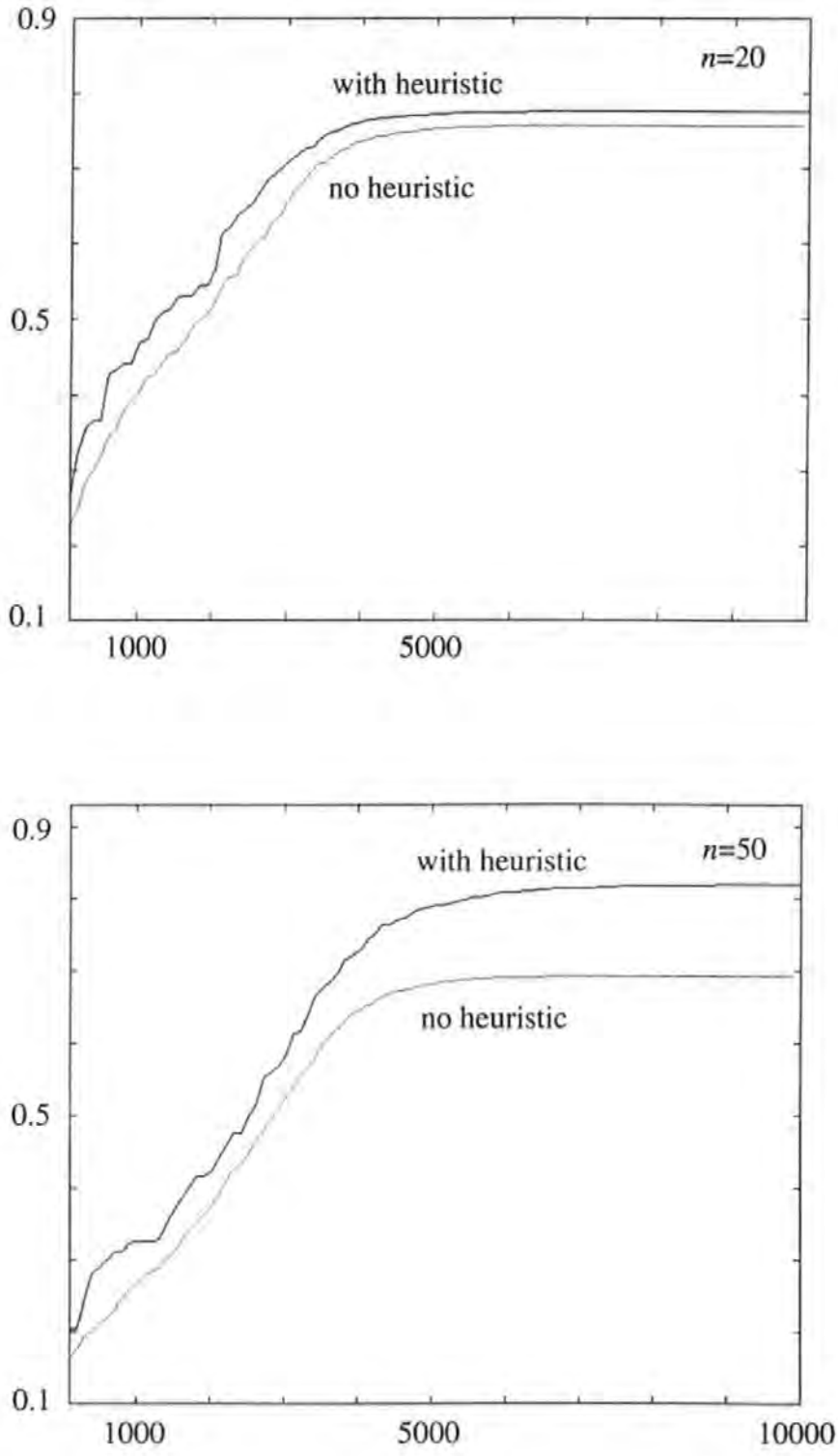


Fig. 4.8. Performance of an ant colony search model on the bump problem. The two graphs show an ant colony model with and without heuristic rules.

CHAPTER 5

Feasibility Search for Problems with Implicitly Defined Constraints

In this chapter we assume that the constraints are implicitly defined (i.e., "black box" representation) or that they are so complex that the feasible region cannot be readily derived explicitly as in chapter 3. The main objective will be to "outline" the feasible region by a population of samples. The developed framework should work in convex as well as non-convex feasible regions. Once the various feasible subregions are found they can be passed to the engineering designer for evaluation.

5.1. Low Discrepancy Sequences

If we want to guarantee a uniform sampling of the search space we can use a grid sampling. The trouble with the grid sampling is that one has to decide *in advance* how fine it should be. One is then committed to completing all the points. With a grid it is not convenient to sample until some convergence or termination criterion is met. One might ask if there is not some intermediate scheme, i.e., some way to pick sample points "at random", yet spread out in some self-avoiding way, avoiding the chance clustering than occurs with uniformly random points. So the question is: Is there any way to sample better than uncorrelated, random samples?

The answer to the above question is "yes". Sequences of n-tuples that fill n-space more uniformly than uncorrelated random points are called *quasi-random* or *low-discrepancy* sequences. A conceptually simple example is the Halton sequence [Halton, 60]. In one dimension the jth number H_j in the sequence is obtained by the following steps: (a) Write j as a number in base b, where b is some prime. (For example, j=17 in base b=3 is 122.) (b) Reverse the digits and put a radix point (i.e., a decimal point base b) in front of the sequence. (In the example we get 0.221 base 3.) The result is H_j. To get a sequence of n-tuples in n-space, we make each component a Halton sequence with a different prime base b. Typically, the first n primes are used.

It is not hard to see how Halton's sequence works: Every time the number of digits of j increases by one place, j's digit-reversed fraction becomes a factor of b finer-meshed. Thus the process is one of filling in all the points on a sequence of finer and finer Cartesian grids and in a kind of maximally spread-out order on each grid (since, e.g., the most rapidly changing digit in j controls the *most* significant digit of the fraction).

Other ways of generating low-discrepancy sequences have been suggested by Sobol [Sobol, 67], Niederreiter [Niederreiter *et. al.*, 94], and others. Bratley and Fox [Bratley *et. al.*, 88] provide a good review and references. In our work we will adopt a particularly efficient variant of Sobol's sequence proposed by Antonov and Saleev [Antonov *et.. al.*, 79].

The Sobol's sequence generates numbers between zero and one directly as binary fractions of length w bits, from a set of w special binary fractions, V_i, i=1,2,...,w, called *direction numbers* [Press *et.al.*, 92]. In Sobol's original method, the jth number X_j is generated by XORing (bitwise exclusive OR) together the set of V_i's satisfying the criterion on i, "the ith

bit of j is nonzero." As j increments, in other words, different ones of the V_i's flash in and out of X_j on different time scales. V_1 alternates between being present and absent most quickly, while V_k goes from present to absent (or vice versa) only every 2^{k-1} steps.

Antonov and Saleev's contribution was to show that instead of using the bits of the integer j to select direction numbers, one could just as well use the bits of the Gray code of j, $G(j)$. Now $G(j)$ and $G(j+1)$ differ in exactly one bit position, namely in the position of the rightmost zero bit in the binary representation of j (adding a leading zero to j if necessary). A consequence is that the j+1st Sobol-Antonov-Saleev number can be obtained from the jth by XORing it with a single V_i, namely with i the position of the rightmost zero bit in j.

Figure 5.1. plots a two dimensional Sobol sequence. One sees that successive points do "know" about the gaps left previously, and keep filling them in, hierarchically.

We have deferred to this point a discussion of how the direction numbers V_i are generated. Each different Sobol sequence (or component of an n-dimensional sequence) is based on a different primitive polynomial over the integers modulo 2, that is, a polynomial whose coefficients are either 0 or 1, and which cannot be factored (using modulo 2 integer arithmetic) into polynomial of lower order. Suppose P is such a polynomial of degree q:

$$P = x^q + a_1 x^{q-1} + a_2 x^{q-2} + \cdots + a_{q-1} + 1 \tag{5.1}$$

Define a sequence of integers M_i by the q-term recurrence relation:

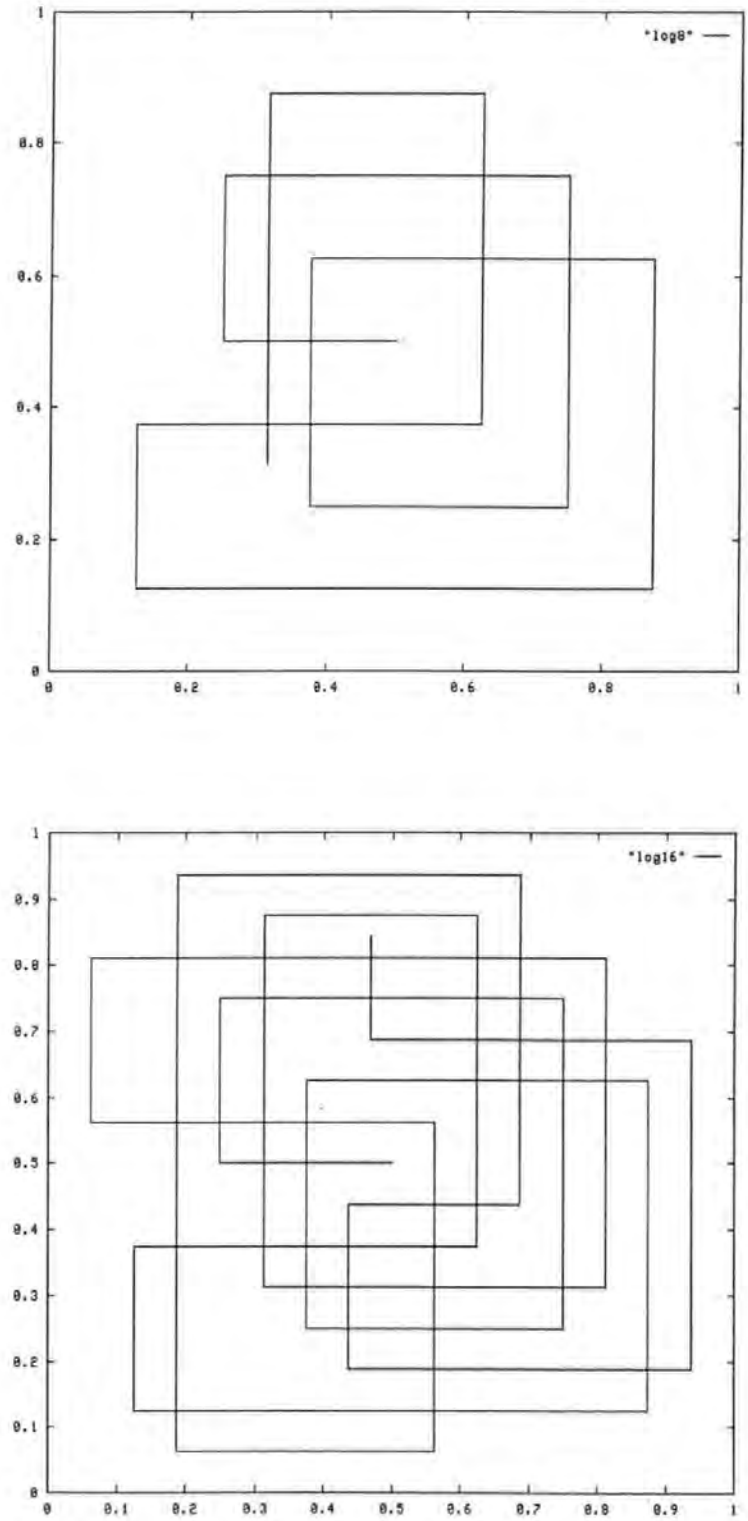


Fig 5.1. "Walk" of a two dimensional Sobol sequence of length 8 and 16, respectively

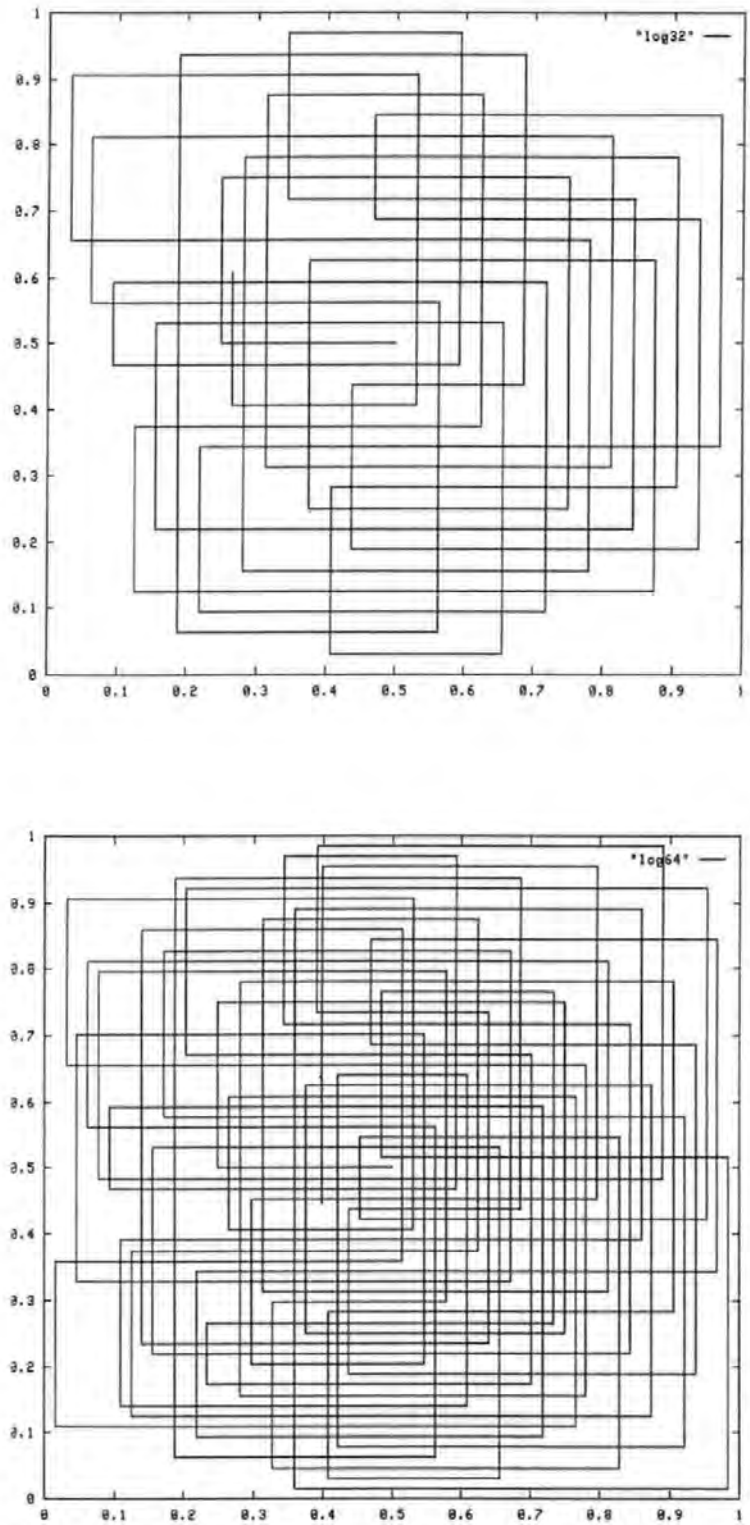


Fig 5.1. (contitnued) "Walk" of a two dimensional Sobol sequence of length 32 and 64, respectively

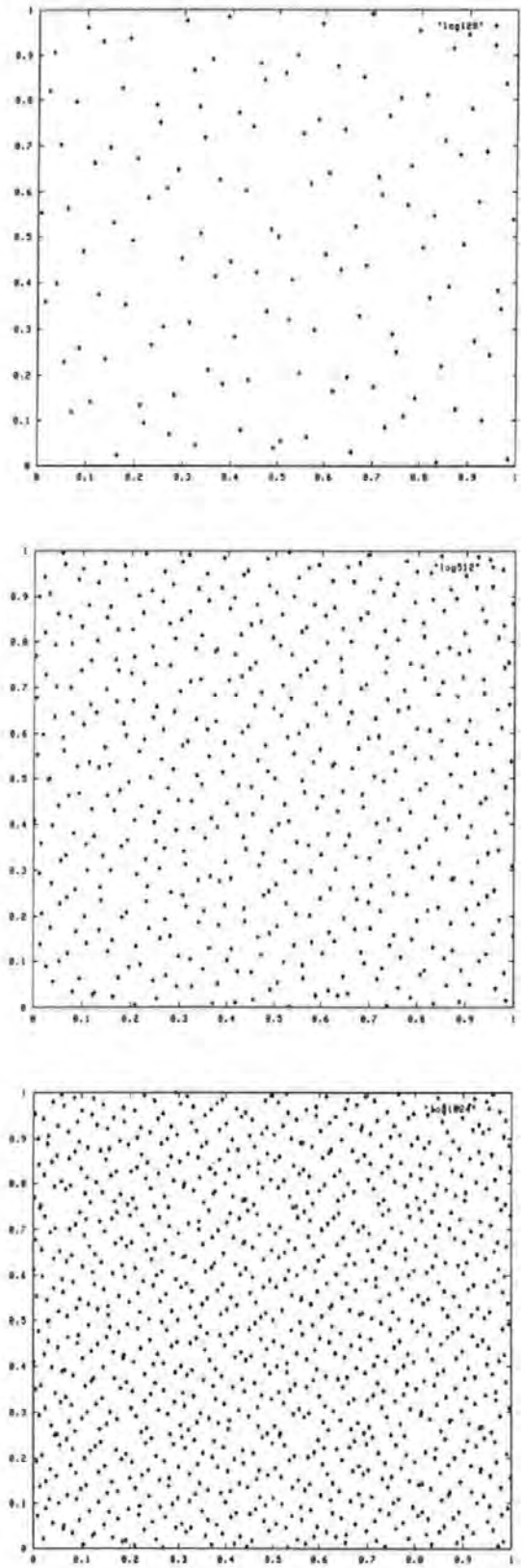


Fig 5.1. (contitnued) A two dimensional Sobol sequence of length 128, 512 and 1024, respectively

$$M_i = 2a_1 M_{i-1} \oplus 2^2 a_2 M_{i-2} \oplus \cdots \oplus 2^{q-1} M_{i-q+1} a_{q-1} \oplus \left(2^q M_{i-q} \oplus M_{i-q}\right) \qquad (5.2)$$

Here bitwise XOR is denoted by $\oplus$. The starting values for this recurrence are that $M_1,...,M_q$ can be arbitrary odd integers less than $2,...,2_q$ respectively. Then the direction numbers V_i are given by:

$$V_i = \frac{M_i}{2^i}, \quad i=1,...,w. \qquad (5.3)$$

5.2. Population-based Identification of the Feasible Region

5.2.1. Implementation Details

In this section we are interested in finding the feasible region using a finite population based search. A straightforward way to outline the feasible region is by uniformly sampling the search space. Although this is an accurate method it is highly computationally expensive. One way to reduce the computational expense but still preserving the guarantee of not missing a feasible "corner" is by utilising a low-discrepancy sequence. Fig. 5.2. shows the feasible region of three arbitrary* 2-dimensional test functions outlined by a Sobol sequence of length 500,000. A C-code implementation of the test functions is included in the Appendix.

In engineering design, however, it is quite usual to be limited to a certain number of cost function calls (e.g., 10,000) depending on the computational cost of the simulation model.

* For the purpose of our investigations the exact definition of the test functions is not so relevant as the "shape" of the feasible region.

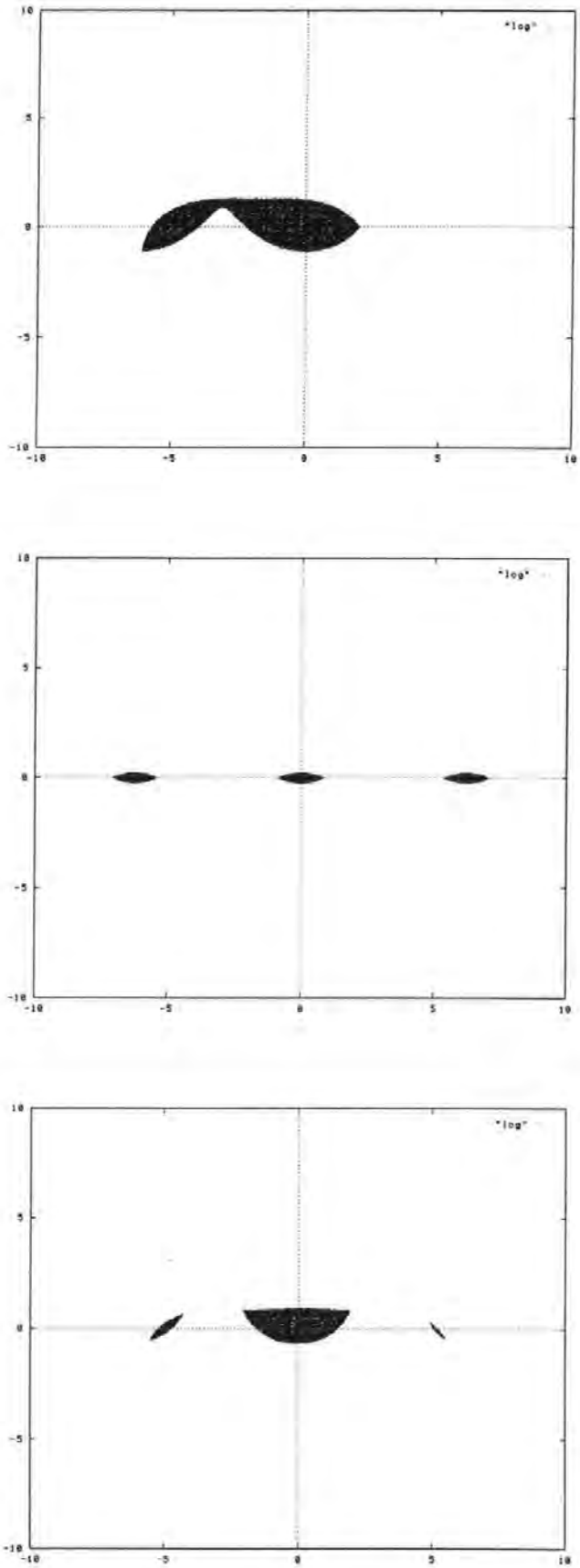


Fig. 5.2. The feasible regions of three test functions as outlined by a two dimensional Sobol sequence of length 500,000. The first test function has a non-convex connected feasible region and the last two test functions have disconnected feasible regions.

Therefore, we need a much faster method for outlining the feasible region, though at the sacrifice of accuracy and guarantee of not missing a feasible region. In this section we develop an adaptive search population based method for identifying the feasible region using a small number of cost function calls. The idea is to modify the dynamic hill climbing to include a low-discrepancy sequence for generating the staring points. Using this approach it is no longer necessary to maintain a database of point neighbourhoods which have already been sampled, since the low-discrepancy sequence "remembers" the regions it has already visited:

(Algorithm 1)

1. Initialize a population of N quasi-random (i.e., from a Sobol sequence) samples.
2. For each of the elements in the population apply a hill-climber for M steps. The cost function is the constraint violation.
3. Repeat step 2 until all individuals are in the feasible region or stuck at local optima.

The above algorithm allows an equal number of hill-climbing steps for each point at each iteration. It is quite possible, however, that the slope of the constraint violation function is different at various points and therefore, the above described population does not converge uniformly on the feasible region (i.e., some points will satisfy the constraint violation better than others). This is not a fundamental deficiency of the proposed algorithm as far as the goals of finding the feasible region is concerned, but will play a significant role when trying to introduce heuristics that reduce the number of cost function calls (i.e., at each iteration it makes more sense to compare points with similar degree of constraint satisfaction). To achieve uniformity of constraint satisfaction during each iteration we modify the above algorithm in the following way:

(Algorithm 2)

1. Initialize a population of N quasi-random (i.e., from a Sobol sequence) samples.
2. Set a constraint violation threshold θ and relax the constraints in the constraint violation cost function by θ%.
3. For each of the elements in the population apply a hill-climber until the modified constraint violation function is satisfied or a local optima is reached (in which case that individual is removed from the population).
4. Tighten the constraint relaxation threshold by a predetermined Δ, i.e., the allowed constrained violation becomes $\theta=\theta-\Delta$%.
5. Goto step 3 until $\theta-\Delta$ becomes zero, in which case the constraints correspond to their original values.

5.2.2. Experiments

The above described algorithm modifies the cost function at each iteration. This approach is quite reminiscent of the inductive search approach described in chapter 2. Fig. 5.3 shows the found feasible region of the test functions from fig. 5.2. The number of cost function calls is 15456, 17336 and 16741, respectively. It turns out that adaptation is capable of reducing the number of calls to the cost function while at the same time presenting an acceptable description of the feasible region. Of course, this cannot be always guaranteed, especially for cost functions with a huge number of local peaks.

Usually, we do not know in advance the exact number of peaks. It is a good idea to start our search with a large population size (i.e., comprehensive exploration of the search space) and then to concentrate the search only to the most promising areas (i.e., exploitation). This approach is only valid if the number of peaks is significantly less than

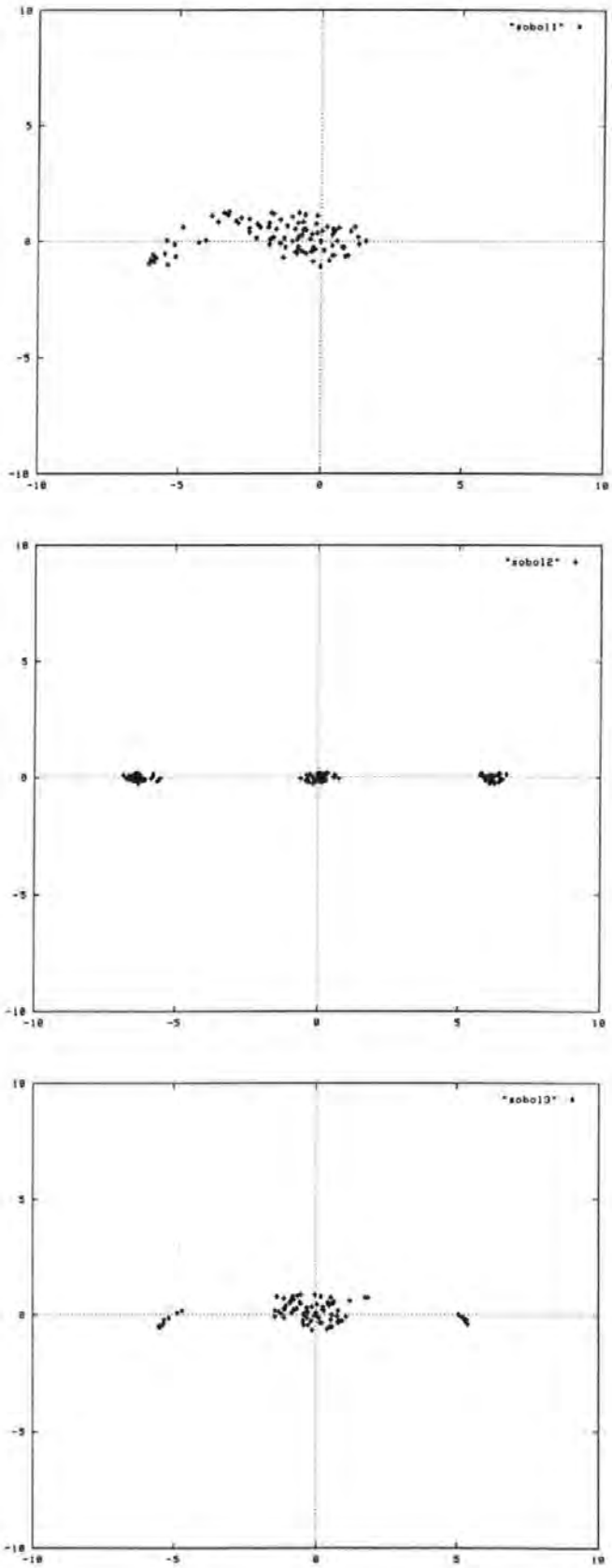


Fig. 5.3. The feasible region of the three test functions from fig. 5.2 outlined by the modified dynamic hill climbing. The number of calls to the cost function is 15456, 17336, and 16741 respectively.

the population size. If the number of peaks is greater than the population size, then it is obvious that such a heuristic reduction of exploration would result in missing feasible regions. Although such a heuristic does not give us a guarantee of not missing a feasible "corner" of the search space, in engineering design practice it may turn that due to sensitivity issues we are more interested in relatively large feasible areas and do not care too much if our heuristic search algorithm filters out some peaky narrow feasible "islands".

In order to implement the above described goals we first borrow some ideas from immunology.

5.3. An Immunity-based System for Finding the Feasible Region

5.3.1. Ideas from Immunology

The practice of vaccination significantly predates some understanding of the immune system. More than a century has passed since Pasteur developed his rabies vaccine, which prevents the otherwise fatal disease. In the nineteen-fifties, about a century after Darwin's "Origin of Species", MacFarlane Burnett proposed the "clonal selection" theory of B-cell response. (B-cells are an important part of the immune system response.) B-cell clones expand through a proliferation of those cells whose surface immunoglobulins bind to the invading antigen (fig. 5.4). The information content of the genome is not large enough to be able to mount an "instructive", genetically pre-programmed immune response. The number of possible antigens is simply too large and unpredictable, and the pathogens evolve much faster then their host species and therefore, can generate ever novel "molecular surprises" for their hosts. The immune system is thus *self-organising.*

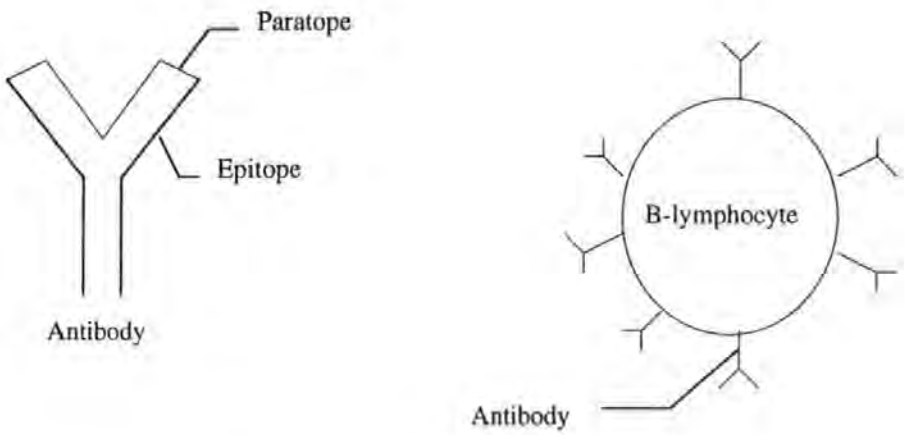


Fig. 5.4. Basic elements of the immune system model. The paratope binds to the surface of invading antigens. The degree of matching that surface corresponds to the degree of recognition.

The mechanism of generating self-defining molecular identity implies that the immune system has to "learn" to distinguish "*self*" from "*non-self*". There are different hypothesis attempting to describe the process of self-identification. MacFarlane Burnett assumes that self-identification is accomplished by a process of *clonal deletion*, during neonatal development, of all those B-cells and T-cells that are *self-reactive*. The clonal selection theory has a counter part: The *idiotypic network theory* of Jerne (who, like Burnett, won a Nobel prize). According to this theory, *antibodies* themselves *become antigens* by carrying *epitopes* that are *antigenic*, and which therefore stimulate other B-cell clones, which in turn carry epitopes, which stimulate other clones, and so on. The result is an "idiotypic network" of B-cells and antibodies that bind to each other and which stimulate and inhibit each other.

5.3.2. Implementation Details.

In this section we utilise the idea of idiotypic interactions and apply it to the population of our search algorithm (section 5.2, Algorithm 2). The goal is to adaptively reduce the number of similar samples. The employed heuristic is that samples which are close to each other (in an Euclidean distance measure) most probably will lead to the same feasible region or peak and therefore, we can delete all similar samples from the population but one. For other applications of immunity-based systems to search and optimisation the reader is referred to [Smith *et. al.*, 93] and [Forrest *et. al.*, 93].

More formally, the idea is as follows. At each iteration of the algorithm described in section 5.2 we calculate the Euclidean distance between all pairs of samples. Then for all pairs that are closer than a given threshold we randomly delete one of the samples from the population. In terms of our immunity-based metaphor, the distance between two samples is analogous to the degree of recognition between two antibodies. The resulting algorithm can

also be viewed as a stochastic clustering technique, in which we are only interested in the center of the clusters.

Under the assumption that similar samples eventually lead to the same feasible region or peak, the above described heuristic reduces the overall number of cost function calls without sacrificing quality of obtained results. The modified algorithm is as follows:

(Algorithm 3)

1. Initialize a population of N quasi-random (i.e., from a Sobol sequence) samples.
2. Set a constraint violation threshold θ and relax the constraints in the constraint violation cost function by θ%.
3. For each of the elements in the population apply a hill-climber until the modified constraint violation function is satisfied or a local optima is reached (in which case that individual is removed from the population).
4. For all pairs of samples that are closer to each other than a given threshold τ delete randomly one of them with probability ρ, $0 \leq \rho \leq 1$. (We have used $\rho=1$.)
5. Reduce the constraint relaxation threshold by a predetermined Δ.
6. Goto step 3 until $\theta-\Delta$ becomes zero.

5.3.3. Experiments.

Fig 5.5. shows the results of the immunity-based population search starting with the same number of initial samples as in the previous experiment (fig. 5.3) on the three test functions from fig. 5.2. The number of cost function calls is reduced considerably to 8108, 9113 and 9009 respectively. The qualitative nature of the feasible region is not sacrificed which is due to the validity of the assumption for the particular tested functions.

It is also possible, however, that the above assumption is not valid. In such case the presented algorithm will fail to faithfully represent the feasible region. This can easily happen for heavily constrained problems where the feasible regions are small and randomly scattered around the design space. Chapter 6 presents one such problem.

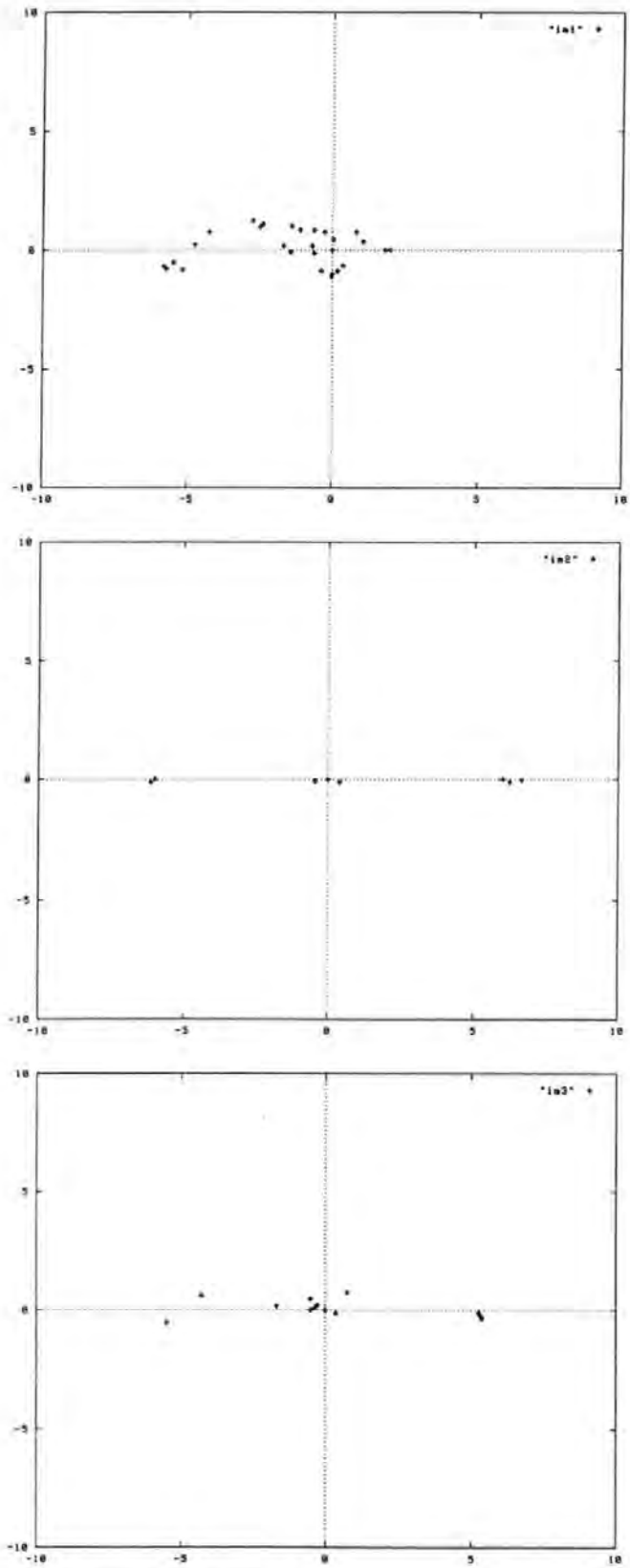


Fig. 5.5. The feasible region of the three test functions from fig. 5.2. outlined by the immunity-based search. The number of calls to the cost functiuons is 8108, 9113, and 9009 respectively.

CHAPTER 6

Feasibility Search for Heavily Constrained Problems

6.1. Heavily Constrained Engineering Design Problems

In this chapter we deal with a heavily constrained optimisation problem from the aircraft design domain. The chosen design problem is realistically complex; its globally optimum solution is not known or readily determined due to the large size of the search space and noise. Most of the desired performance criteria are defined as constraints which are implicitly implemented into a simulation model. Their values can only be accessed after the simulation has completed.

6.1.1. Preliminary Aircraft Design

The application of AI and advanced software techniques to engineering design is resulting in the development of new software tools for the design of aircraft [Dixon, 86]. The research in this chapter further contributes to this field and aims at integrating the adaptive search technology for feasibility search.

The first step in constructing methods for aircraft design is to consider the general nature of the problem of engineering design [Bouchard *et. al.*, 88]. Fundamentally, engineering design is the translation of some set of functional desires into a set of instructions that can be used to "construct" an object that satisfies those desires. In practice, the design process

typically generates a largely geometrical description, known as configuration. In this less complete but more common view of design, the configuration represents an implicit set of instructions for constructing the object.

Parametric design is a subtask of design. The design concept, which is the general type and arrangement of the object being designed is the starting point of parametric design. In addition, the desires to be satisfied form a set of constraint or objective requirements. The design for an aircraft, for example, requires specifying details: Does the aircraft have wings? If so, what type are they, how big are they, and where are they located? When parametric design starts, many of these decisions have already been made; the results are expressed in the design concept (usually implemented as a simulation model). This concept might specify that the aircraft has wings made of aluminium and a jet engine buried in the fuselage. The object of parametric design is to produce a specific design from the family of designs implied from the design concept. This entails answering questions like: How big should the wing and engine be for a minimum-weight aircraft that meets the range requirement?

A configuration can be specified by a set of symbolic and numerical characteristics that define the objects and relations which comprise the configuration. Selecting characteristics that can be specified independently produces a set of design variables. In parametric design, the design variables define an instance of the design concept being examined. Examples of aircraft design variables are tail area, tail location, engine size, etc. These variables provide the means by which the design can be optimised subject to the design requirements.

Because paramteric design starts from a design concept, it avoids much of the synthesis, reasoning by analogy, and common sense reasoning which are required in other types of design activities. This makes it a highly suitable candidate for automation and is one of the reasons that the emerging design tools have concentrated on this area of design.

6.1.2. The "Hotol" Project by British Aerospace Plc.

The design domain of the Hotol aircraft involves the preliminary parametric airframe design and definition of a flight trajectory for an air-launched winged rocket that will achieve orbit before returning to atmosphere for a conventional landing. The trajectory consists of a pull-up from air launch at 9000m altitude and Mach 0.8 at a constant incidence, followed by a zero incidence ascent. The main engine cut-off (MECO) window to aim for is 90km altitude, approximately 7500m/s and a small climb angle to put it into an approximately 50x300 mile elliptic transfer orbit.

The fuselage is a cylinder with spherical cap. The wing is straight tapered. The mass of the fin is accounted for, but no aerodynamics are modelled. Due to the geometry of tanks, allowance for wing carry-through structure, payload bay, guidance and systems, the volume of fuselage available for fuel is less than might be expected (which is accounted for in the simulation model). The fuselage mass is assumed to be composed of two approximately equal components, one proportional to the surface area, one to volume. The wing is calculated to a NASA formula, assuming a maximum load factor of 1.5 at full load (i.e., in the pull-up), with a reserve factor of 1.4. The fin is based on shuttle data. The engine uses current knowledge of T/W as a function of scale. The payload is specifiable: the vehicle on which our simulation model is based is designed to a 7000kg payload.

Lift is calculated by the proprietary DATCOM formula, with Clmax conservatively assumed to be 1.0. No Mach effects are represented as lift is only significant in the pull-up. Drag is made up of fuselage, wing, base and lift-dependent components. The fuselage and wing have skin friction components calculated by the empirical Prandtl-Schlichting formula. Wing wave drag (supersonic) is based on the ESDU method derived by British Aerospace at Warton. Fuselage wave drag is crude, assuming ram drag (i.e., flow brought to a dead stop) on the nose cap. This is pessimistic below Mach 5, but realistic at hypersonic speeds. Base drag applies to the area not covered by engine nozzle, and uses empirical data.

The engines are fitted into the base area such that the nozzles do not protrude beyond the fuselage cross-section (i.e., no shroud). As the engine throat area is proportional to scale factor, the expansion ratio (nozzle area/throat area) follows from this criterion. Up to 7 engines may be fitted; beyond that the geometric packing becomes complicated. Thrust is calculated directly from specific impulse and fuel flow, which is proportional to engine scale.

The design concept as modelled by the simulation code is schematically shown in fig. 6.1. The independent variables of the design concept include:

ALPHA: Incidence during pull-up (in degrees)

GAMMA: Climb angle at the end of pull-up (in degrees). The trajectory and conditions at MECO (Main Engine Cut-Off) are very sensitive to ALPHA and GAMMA.

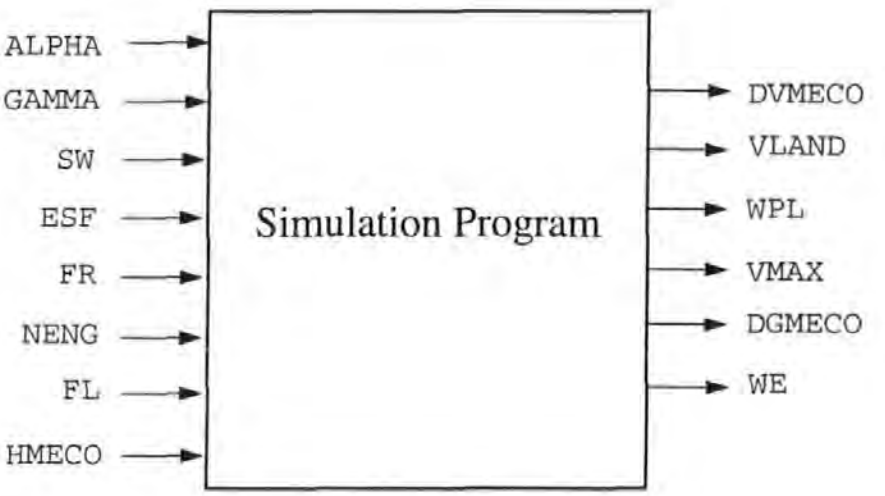


Fig. 6.1. A schematic representation of the Hotol parametric design concept.

SW: Gross wing area. This means the area of the wing considered projected to the vehicle centreline.

ESF: Engine scale factor. This is relative to a 1000kN engine.

FR: Fuselage fineness ratio (length/diameter).

NENG: Number of engines.

FL: Fuselage length.

HMECO: Height at MECO. This relates to the required orbit.

There are several constraints which are to be met:

DVMECO: Speed excess at main engine cut-off (m/s). This is the excess over that required to achieve the specified orbit. This should be obviously zero.

VLAND: Landing speed (m/s). This should be about 77 m/s.

WPL: Achieved payload weight. Nominal value of 7000 kg.

VMAX: Maximum speed in ascent (m/s). This is a measure of the kinetic heating during ascent, and also affects the loading on the structure. A typical compromise limit may be 260 m/s but this may not be possible to achieve without modulating thrust.

DGMECO: Excess in climb angle over the required for orbit at MECO. This should be obviously zero.

Initially a tolerance of 1% on the nominal values is accepted as satisfactory. This would give margins of about ±70, ±1, ±70, ±2, ±0.01 respectively.

The overall objective of the design is to minimise the empty weight of the vehicle (WE), based on the specified geometry.

6.1.3. Current Solution Procedure Provided by British Aerospace Plc.

Using current experience and deep knowledge of the simulation model British Aerospace provided a manual procedure for designing a Hotol airframe. At first, the problem is divided into two subproblems: (1) definition of the vehicle and (2) achieving the required orbit. The reason behind this division is that trajectory optimisation is a difficult, but well understood problem. The design procedure is as follows:

1. Guess the fuselage length (FL): 40m is at least sufficient.
2. Adjust the fineness ration (FR): For aerodynamic drag considerations this value should be high. At the same time, for structural efficiency considerations, this value should be 1. A good compromise is somewhere between (i.e., 5).
3. Taking into consideration takeoff mass and landing speed guess the gross wing area (SW).
4. Guess number of engines (NENG). No heuristic is provided at this stage.
5. Adjust the engine scale factor (ESF) to give T/W ~ 1.5.
6. HMECO has to be in the orbit i.e., between apogee and perigee and below major semiaxis of ellipse.
7. Play with ALPHA and GAMMA to meet trajectory constraints.

The first five steps from the above design procedure define the vehicle and the last two steps optimise on the trajectory. It will be shown later in this chapter that such subdivision of the problem is not efficient since the definition of the vehicle, to a great extent, predetermines the success of the trajectory optimisation procedure. It is well justified, therefore, to consider both stages in parallel.

Using the above defined design procedure British Aerospace have designed the following airframe:

ALPHA:	19.0
GAMMA:	44.58
SW:	322
ESF:	1.20
FR:	4.55
NENG:	4
FL:	40.10
HMECO:	90000.00

which has the following simulation results:

DVMECO:	-78.6001
VLAND:	65.3845
WPL:	20050.1
VMAX:	229.535
DGMECO:	-0.57893
WE:	28104.0

A quick look at the simulation results reveals that the constraints (as initially defined) are not satisfied. Two major questions arise: (1) Can we find a solution that satisfies the defined constraints, and (2) What is the effect of constraint relaxation on the difficulty of the search problem? The answer to the first question is largely unknown *a priori* and it is suggested that the relaxation of the constraints will make the search for a feasible region easier as the feasible region itself will become relatively larger.

6.2. Constraint Satisfaction in Heavily Constrained Problems

The problem provided by British Aerospace Plc. is a constrained optimisation problem with an extremely difficult *feasibility part* (i.e., finding a feasible point). It has seven real and one discrete design variables and five real valued non-linear non-explicit (i.e., integrated into a simulation code) constraints. The problem is of non-convex nature as the simulation code often results in errors for which the simulation outcome is not defined. The problem is also noisy due to the numerical simulation.

Problems with a difficult feasibility part are often referred to as *heavily constrained* problems. For such problems if the feasible region is disconnected (as will prove to be the case with the Hotol problem) the constrained optimisation part reduces to finding the set of feasible regions.

6.2.1. Definition of a Constraint Violation Function

It is well known that an appropriate definition of a constraint violation function is of paramount importance to the success/failure of any search algorithm. We have found the following definition very useful [Bilchev and Parmee, 95b]:

$$F(x) = \begin{cases} \sum_{i=1}^{5} cnstr\_viol_i(x)^2 & \text{if normal simulation termination} \\ C & \text{otherwise} \end{cases} \qquad (6.1)$$

$$cnstr\_viol_i(x) = \begin{cases} \dfrac{c_i(x) - u_i}{u_i - l_i} & \text{if } c_i > u_i \\ \dfrac{l_i - c_i(x)}{u_i - l_i} & \text{if } c_i < l_i \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

where C is a large constant which penalises errors in the simulation program, and l_i and u_i are lower and upper bounds of the feasible region as defined by the problem.

The constraint violation function thus defined assumes (1) equal importance of all constraints, and (2) equal difficulty in satisfying them. If this is to be changed and we want to attribute different weights to the various constraints, we can change the constraint violation function to:

$$F(x) = \begin{cases} \sum_{i=1}^{5} W_i \cdot cnstr\_viol_i(x)^2 & \text{if normal simulation termination} \\ C & \text{otherwise} \end{cases} \tag{6.3}$$

where W_i represent our knowledge of constraint satisfaction difficulty and/or our constraint satisfaction preference. More difficult constraints should have higher weights as well as the more important to satisfy (from an engineering design point of view) constraints. Constraint relaxation can be controlled by the lower and upper bounds of the feasible region (l_i and u_i) and reflects the notion of softness/hardness in the constraint definitions.

6.2.2. Experiments with Various Optimisers

6.2.2.1. Application of Direct Pattern Search of Hooke and Jeeves

The *direct pattern search* of Hooke and Jeeves [Hooke and Jeeves, 61], originally devised as an automatic experimental strategy, is nowadays much more widely used as a numerical parameter optimisation procedure. The method is characterised by two types of move. At each iteration there is an *exploratory move*, which represents a simplified Gauss-Seidel variation with one discrete step per co-ordinate direction. No line searches are made. On the assumption that the line joining the first and the last points of the exploratory move represents an especially favourable direction, an extrapolation is made along it (*pattern move*) before the variables are varied again individually. The extrapolations do not necessarily lead to an improvement in the objective function. The success of the iteration is only checked after the following exploratory move. The length of the pattern step is thereby increased each time, while the optimal search direction only changes gradually. This pays off to most advantage where there are narrow valleys, provided they are not sharply bent. The extrapolation step s follows, in an approximate way, the gradient trajectory. However, the limitation of the trial steps to co-ordinate directions can also lead to premature termination.

A proof of convergence of the direct search of Hooke and Jeeves has been derived by Cea [Cea, 71]; it is valid under the condition that the objective function is strictly convex and continuously differentiable.

However, the design space S of the Hotol problem is of non-convex nature. The objective function is often not well defined over the simple search boundaries l_i and u_i. This is due to the fact that the simulation often results in an error return code where the constraints do not have a meaningful value to be used as gradient information. Moreover, the objective

function is noisy (due to numerical integration procedures in the simulation code) and has a finite number of discontinuities. Therefore, the proof of convergence of the direct search method of Hooke and Jeeves (as well as the proof of any other method assuming convex, continuously differentiable functions) is not valid.

However, in a close proximity to local optima it is most likely that the objective function is convex and continuously differentiable. Therefore, it is worth trying the direct search method of Hooke and Jeeves around such local optima.

The algorithm of Hooke and Jeeves with improvements due to Bell and Pike [Bell and Pike, 69], and Smith [Smith, 69] is described in Appendix C. We apply it to the Hotol design problem starting from the solution provided by BAe. The constraint satisfaction regions are slightly relaxed from the original formulation and express the notion of *acceptable* designs:

DVMECO: $\in [-20, 500]$
VLAND: $\in [0, 77]$
WPL: $\in [7000, 3000]$
VMAX: $\in [0, 300]$
DGMECO: $\in [-0.01, 0.01]$

The cost function used is the constraint violation with equal weights. The initial step sizes for the algorithm are defined as follows:

H_{ALPHA}: 1.0
H_{GAMMA}: 1.0
H_{SW}: 10.0
H_{ESF}: 1.0
H_{FR}: 1.0

H_{NENG}:	N/A (Number of engines is kept fixed to 4)
H_{FL}:	1.0
H_{HMECO}:	100.0

where the number of engines is fixed equal to the number of engines of the starting point. After 467 cost function evaluations the algorithm converges to:

ALPHA:	19.0
GAMMA:	44.58
SW:	322
ESF:	1.20
FR:	4.55
NENG:	4
FL:	40.10
HMECO:	91528.38

where:

DVMECO:	3118.59
VLAND:	65.3845
WPL:	-11471.5
VMAX:	229.535
DGMECO:	-0.0099

WE:	59625.65

It can be seen from the simulation result that three of the constraints are satisfied (namely, VLAND, VMAX, and DGMECO). This allows us to increase the weights of the constraint violation of the other two constraints in order to attempt to drive the search process into satisfying them as well. We start again from the same initial point, but this time with the following weights:

W_{DVMECO}: 10

W_{VLAND}: 1

W_{WPL}: 10

W_{VMAX}: 1

W_{DGMECO}: 1

After 452 cost function evaluations the following the algorithm converges at:

ALPHA: 19.0

GAMMA: 44.58

SW: 322

ESF: 1.20

FR: 4.55

NENG: 4

FL: 40.10

HMECO: 90164.24

where:

DVMECO: 499.9

VLAND: 65.3845

WPL: 12568.1

VMAX: 229.535

DGMECO: -0.54731

WE: 35586

In conclusion, the algorithm of Hooke and Jeeves is capable of slightly improving on the results when started at a near optimum point, but it is the nature of the problem that does not allow a local hill climber to find a feasible solution. It seems that the vehicle itself is not appropriately defined to allow a feasible trajectory performance when tuning only the trajectory parameters. Therefore, the problem requires a simultaneous design (search) along both the vehicle definition and trajectory optimisation. Genetic algorithms are quite

suitable for such application as they are capable of evolving a population of potential candidate designs each of which can encode both vehicle and orbit parameters.

6.2.2.2. Application of the Genetic Algorithm

Designed to search irregular, poorly understood spaces, genetic algorithms (GAs) are general purpose algorithms developed by Holland [Holland, 75] with precursors suggested by Bledsoe [Bledsoe, 61] and others. Holland's hopes were to develop powerful, broadly applicable techniques, to provide a means to attack problems resistant to other known methods. Inspired by the example of population genetics, genetic search is population based, and proceeds over a number of generations. The criterion of "survival of the fittest" provides evolutionary pressure for populations to develop increasingly fit individuals. Although there are many variants, the basic mechanism of a GA consists of:

1. Evaluation of individual fitness and formation of gene pool.
2. Recombination and mutation.

Individuals resulting from these operations form the members of the next generation, and the process is iterated until the system ceases to improve.

The most obvious factors that affect the performance are the parameter settings for population size, crossover rate, and mutation rate. The most influential factor, however, is the choice of an encoding scheme or representation. The reason is that a proper choice of representation can significantly help the GA to converge to the global optima.

For optimisation of functions over continuous domains it is sometimes more convenient to select a floating-point representation because it is the natural base for expressing real

valued parameters and it facilitates interfacing with other algorithms (e.g. standard numerical analysis algorithms, regression analysis, etc.). If there are mixed discrete and real design variables, we can mix the representation as well, i.e., the discrete variables will have a binary representation and the real variables will have a floating-point representation.

To apply the GA to the Hotol problem, we use a mixed representation and a population size of 100 chromosomes. The main operator is a dynamic mutation which reduces the perturbation effect on the offspring chromosome as the generation number increases. This guarantees higher precision search at the end of the evolution. Crossover plays a secondary role for floating point representations, since it is only limited to cross at the boundary of a design variable. Each generation produces 100 new chromosomes which are placed in a genetic pool with their parents. A roulette wheel selection is used to select the best 100 which will be the parents of the next generation.

Special attention is devoted to the design of the fitness function. We utilise the constraint violation function as a fitness function in order to drive evolution towards a feasible solution. At this stage it seems that a proper *a priori* selection of the weights W_i can successfully guide the search towards a feasible region. However, the problem here is that we do not have that *a priori* knowledge. One way to overcome this problem is to adopt an adaptive fitness function, implemented by adaptive weights. If a particular constraint is relatively harder to satisfy, then its weight is increased and vice versa. At the beginning of the evolution we set all weights to be 1's.

The results of the application of the GA to the Hotol problem (averaged over 10 independent runs) are shown in fig. 6.2. The shown fitness function is w.r.t. all weights being equal to 1. Some of the best found solutions look like:

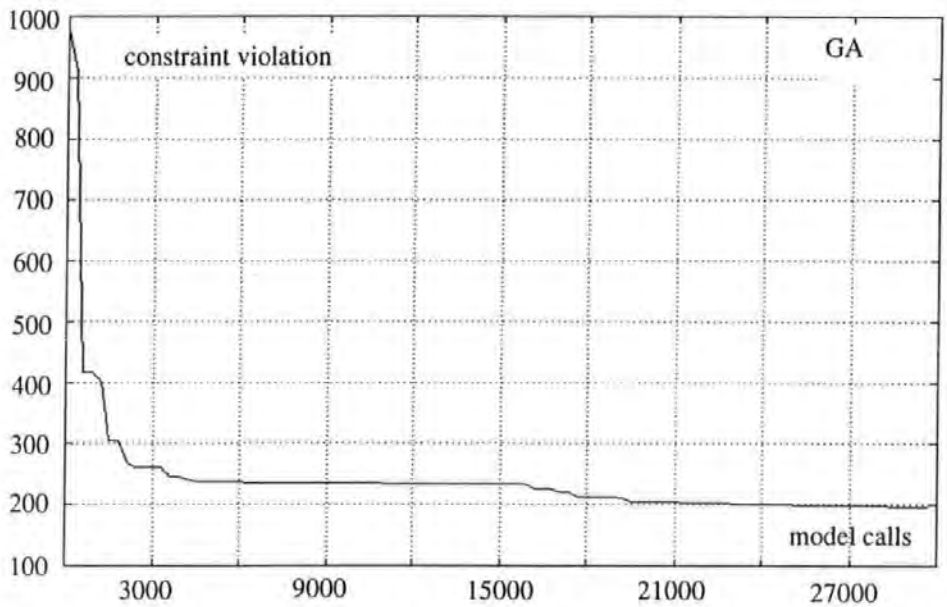


Fig. 6.2. Result from running a GA with the constraint violation calculated as the Euclidean distance from the feasible region.

Design #1:

ALPHA:	16.3336
GAMMA:	51.2857
SW:	397.597
ESF:	0.94375
FR:	9.73702
NENG:	6
FL:	58.4827
HMECO:	74007.89

where:

DVMECO:	104.02
VLAND:	57.732
WPL:	6826.8
VMAX:	238.86
DGMECO:	0.4102

WE:	39530.4

Design #2:

ALPHA:	22.659
GAMMA:	44.249
SW:	168.77
ESF:	1.6825
FR:	5.4751
NENG:	2
FL:	43.469
HMECO:	75154.5

where:

DVMECO:	1240.05
VLAND:	83.202
WPL:	7107.04
VMAX:	251.604
DGMECO:	-0.2622
WE:	33762.8

It is obvious that a significant improvement over the manual design procedure and the classical Hooke and Jeeves algorithm has been achieved by the GA. However, in terms of satisfying the design goals, it still remains to find a feasible solution. In the next section we use the idea of interfacing the population evolved by the GA with our ant colony search model in the hope of finding a feasible solution.

6.2.2.3. Application of the Ant Colony Search Model

In this section we define a hybrid search framework that consists of a GA utilised as a pre-processor for allocating promising feasibility areas of the search space followed by an ant colony search starting from the points found by the GA. The overall structure of the hybrid search is shown in fig. 6.3.

Parmee [Parmee *et.al.*, 94][Parmee and Denham, 94][Parmee, 95b] has shown that a GA with modified selection mechanism and variable mutation is capable of allocating various good design clusters. The implicit cluster information can then be passed to the Engineering Designer (ED) who according to his expertise selects points for further refinement by the ant colony. The ED can either be a human designer or alternatively can be implemented as a filter that passes all designs that are close to feasibility. The ant

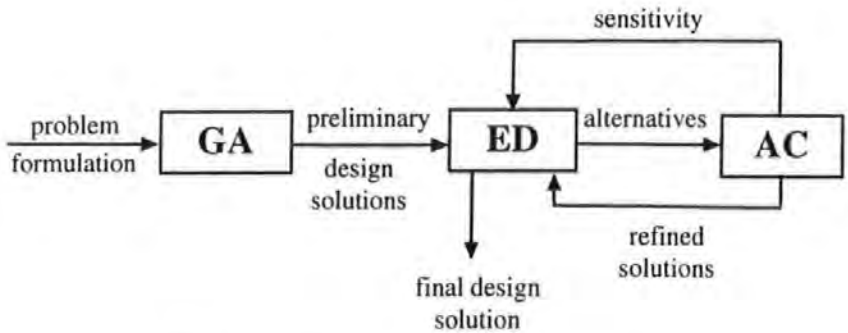


Fig. 6.3. A hybrid search framework.

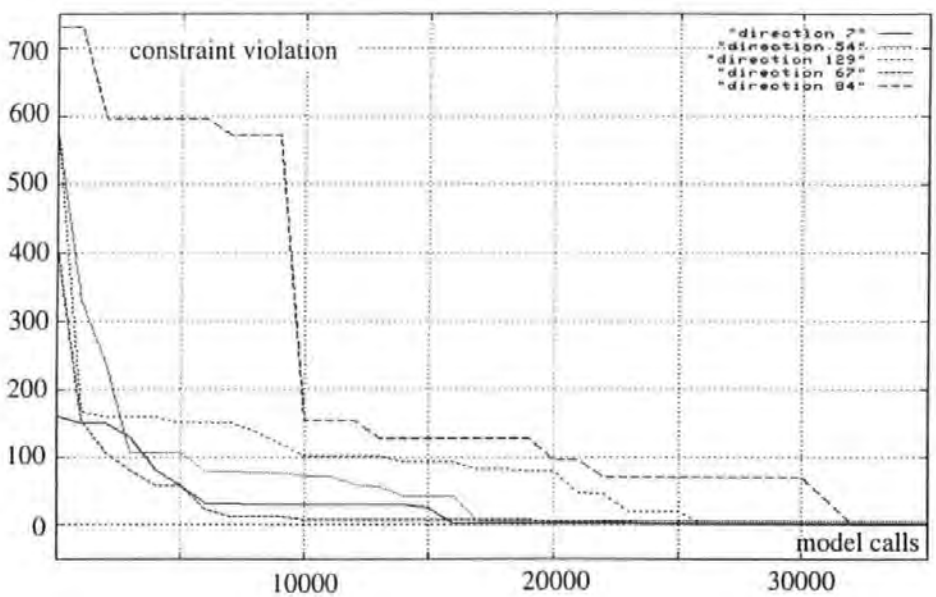


Fig. 6.4. Result from running the ant colony search starting from the best points found by a GA.

colony search is selected because it is a robust multi-modal search technique relying on multi-agent co-operation in order to distribute search in the most promising areas as has been seen in chapter 4. Apart from locally tuned solutions the ant colony can also return an estimate of the sensitivity of a design solution (sect. 6.3).

The proposed hybrid search framework is capable of finding numerous acceptable solutions as shown in fig. 6.4. The GA is run for about 5,000 fitness function evaluations followed by an ant colony search of 1,000 ants for 30 cycles. This makes a total of 35,000 fitness function evaluations which takes approximately 10 hours on a SPARC 10 station. The increased computational cost pays off as several feasible solutions can now be successfully identified. Some of the best solutions are:

Design #1:

ALPHA:	16.20186
GAMMA:	52.20023
SW:	287.6505
ESF:	1.943317
FR:	9.749463
NENG:	1
FL:	55.07114
HMECO:	85094.05

where:

DVMECO:	445.6
VLAND:	57.6
WPL:	7600
VMAX:	238.3
DGMECO:	-0.001

WE: 25819

Design #2:

ALPHA:	24.34185
GAMMA:	51.05346
SW:	231.7679
ESF:	1.721016
FR:	8.089187
NENG:	2
FL:	58.0451
HMECO:	96103.59

where:

DVMECO:	80.2
VLAND:	75.3
WPL:	22025
VMAX:	235.5
DGMECO:	0.001

WE: 23965

In order to accept a feasible solution as a potential design it is crucial to calculate the constraint sensitivity. In the next section we define the constraint sensitivity in a worst-case deterministic setting and an average setting and modify the ant colony search model for the calculation of the sensitivity information.

6.3. Constraint Sensitivity Issues in Heavily Constrained Problems

A sensitivity measure is defined in terms of maximum and average risk of achieving a degraded real design when deviating from the numerically represented design solution. That risk is unavoidable because of the physical impossibility of achieving the exact values of the design variables and the uncertainties in the simulation model itself. It may happen that even when using numerically stable algorithms the found optimal design solution lies within a very sensitive region and a small perturbation in the design variables can lead to an enormous change in the overall design solution [Parmee and Denham, 94]. This is a property of the problem itself and does not depend on the actual optimisation algorithm. So when making decisions the engineering designer, among others, should take into consideration the sensitivity of a given solution.

In this section we present definitions of worst-case and average-case sensitivity and develop a method for calculating it which is applicable to problems with non-explicit objective and constraint functions. The proposed sensitivity measure is also capable of representing design variable interaction.

6.3.1. Definition of Constraint Sensitivity

We are interested in two sensitivity measures reflecting the maximal (worst case) and the average degradation that can be achieved. The degradation is locally defined w.r.t. a particular design point P and should be a function of the distance δ from P (fig. 6.5). (6.4)

The worst case degradation is defined as:

$$\vec{S}_{\max}(\delta) = \max_{(x_1^P - x_1^{P'})^2 + \ldots + (x_n^P - x_n^{P'})^2 = \delta^2} f(P) - f(P') \qquad (6.4)$$

It is a vector with direction pointing to the worst case degradation at a distance δ from P and of absolute value equal to the degradation of the cost function. The projection of $\vec{S}_{max}$ along the design variables co-ordinates is a measure of the contribution of each variable towards the degradation and can be used to estimate the relative sensitivity of the design w.r.t. the individual independent design variables. The calculation of $\vec{S}_{max}$ requires a search at each hypersurface

$$(x_1^P - x_1^{P'})^2 + \ldots + (x_n^P - x_n^{P'})^2 = \delta^2 \tag{6.5}$$

in order to find the maximal design degradation.

Analogously, the average degradation is defined as:

$$S_{aver}(\delta) = \underset{(x_1^P - x_1^{P'})^2 + \ldots + (x_n^P - x_n^{P'})^2 = \delta^2}{aver} f(P) - f(P') \tag{6.6}$$

It is a scalar with value representing the average achieved design degradation at distance δ from P. The calculation of S_{aver} is obvious and involves stochastic sampling at each hypersurface.

6.3.2. An Ant Colony Search for Sensitivity Calculation

We now apply the ant colony search to the sensitivity calculation problem. The only care that must be taken is to design an appropriate fitness assigning model. The algorithm

works as follows: At each generation ants are constrained to search only on the hypersurface:

$$(x_1^P - x_1^{P'})^2 + \ldots + (x_n^P - x_n^{P'})^2 = \delta_t^2 \tag{6.7}$$

where δ_t is the search radius at generation t. At the next generation the search radius is incremented by Δ. The best solutions found at radius δ are propagated into the initial population of the $\delta+\Delta$ search (fig. 6.6). This heuristic is well justified by some *a priori* continuity assumptions.

We calculate the constraint sensitivity information at the two designs discovered by the hybrid search model (section 6.2.2.3). Results are shown in fig 6.7. It can be easily seen that design #2 is more robust w.r.t. constraint sensitivity than design #1. A slight variation of design #1 can easily make it significantly infeasible.

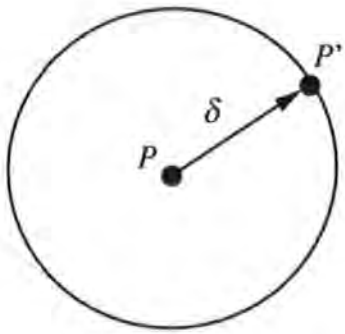


Fig. 6.5. Definition of local design sensitivity at a radius δ around a proposed solution point P.

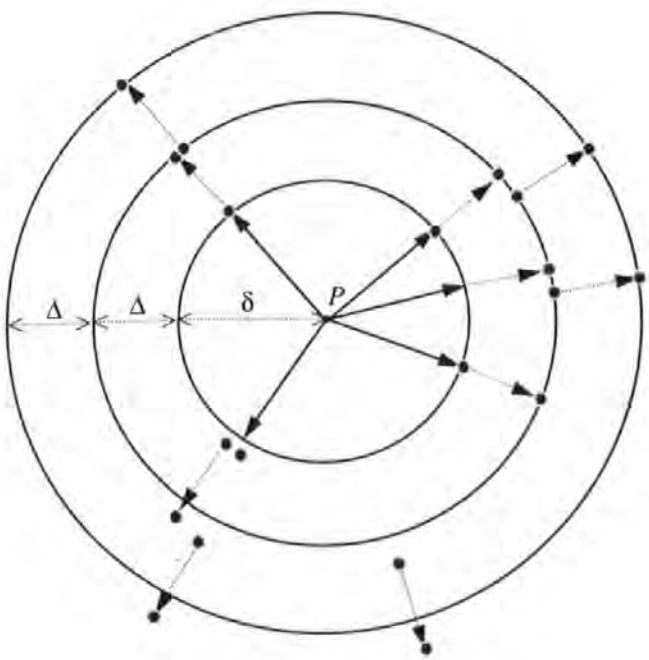
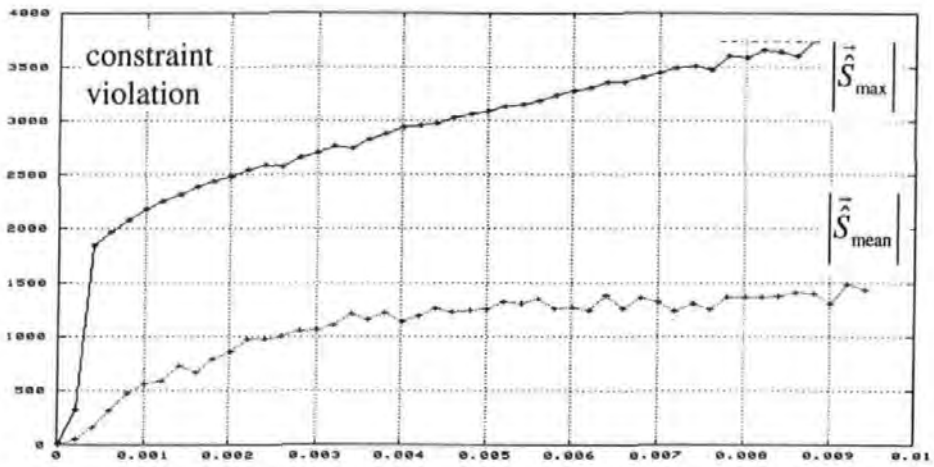
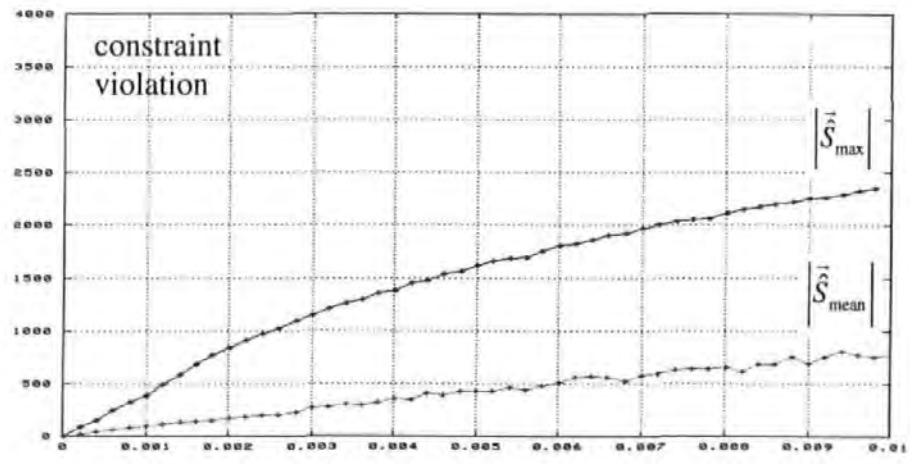


Fig. 6.6. An ant colony search is used to find the maximum degradation of the design found on a hypersphere with radius δ. Then the best solutions are propagated into the initial population of the next (δ+Δ) search problem.



design #1



design #2

Fig. 6.7: Sensitivity calculated at two points of the search space

CHAPTER 7

Discussion and Conclusions

7.1 Discussion

In this dissertation we have investigated the use of constraints to explicate design questions and circumscribe feasible regions. We have examined the process of search and scrutiny within a region. We have viewed constraints as the rules, requirements and relations that are defined within the context of designing.

Constraints are imposed by nature, culture, convention and marketplace. Some are imposed externally, while others are imposed by the designer. Some are the result of higher-level design decisions; some are universal (e.g. gravity, molecular forces, etc.). In this view, to design is to describe constraints and to specify an object that satisfies all these constraints. This was the goal of study of the dissertation, i.e., developing novel adaptive search (extrema finding) methods for specifying objects that satisfy the constraints already defined by a simulation model (i.e., a design concept).

Search in design is quite different from extremal problem solving (i.e., optimization) in mathematical programming. The difference mainly stems from the nature of the design problem itself. Design problems are atypical problems in that they have many solutions. The objective is not to find the solution to a set of design specifications; we find several

solutions out of many alternatives. Stated in other words, in engineering design the goal is not to find *the* solution to *a* problem, but to find *an acceptable* solution to *the* problem.

Another difference is that in the mathematical formulation of extremal problem solving there is no formal difference between easily solvable, explicitly or implicitly defined constraints. In engineering design, however, when we deal with highly complex real world domains, the utilization of *a priori* knowledge or engineering expertise and exploitation of constraint information proves practically to make a great difference in method efficiencies. Therefore, the work presented in this dissertation describes constrained optimization in engineering design viewed as a hierarchy of gradually increasing in complexity problems. Problems in which the constraints can be naturally integrated into the model are generally considered easier to solve than problems where one simulation run returns a set of values which can be combined into a constrained function in many arbitrary ways.

Current results show that the formulation (i.e., the model or the design concept) of the constrained optimization problem greatly effects its difficulty. If it is possible to account for all feasible regions and group them together in one cost function then the search engine views the problem as essentially unconstrained. However, there is no general methodology of handling constraints through the model representation, and therefore, it is natural to expect models in which there are large "holes" of infeasibility.

Quite often the task of the engineering optimizer is hindered by an ill-defined simulation model or model which lacks systematic description of the feasible region thus allowing feasible solutions to be randomly scattered around the search space. As far as engineering design is concerned, it is expected that a closer coupling between the modeling and optimization can significantly improve the achieved results.

Such future research will also have to address more closely the question of reality and models. It is well known from theoretical computer science that the way an object is described determines the set of easy and the set of intractable questions. Therefore, the problem will be how to make such a model which facilitates the search for an answer.

7.2. Summary of Results

One of our main contributions is the development of the ant colony search model as a hybrid optimization framework of co-operating search agents. We showed how to apply it to both discrete (chapter 2) and continuous (chapter 4) problems as well as real world engineering design problems (chapter 6). Experimental results proved that the approach is viable and if it is enhanced with some problem specific knowledge it is also very competitive as compared to existing global optimization techniques.

We have also developed an inductive search approach applicable to both continuous (chapter 2) and discrete (chapter 3) problems. The inductive search has also been successfully integrated into the genetic algorithm model by adding a new layer of dynamically changing fitness function (chapter 3).

In chapter 5 we have extended the dynamic hill climbing paradigm by incorporating a low-discrepancy sequence to generate the starting points and employed analogies from the immune networks to achieve dynamic clustering of the search population. The resulting algorithm maintains the qualitative nature of the feasible region while reducing the number of necessary cost function calls.

7.3. Conclusions

Recent technological advances in computing hardware are offering new ways to extend our problem-solving capabilities. A 200 MHz Pentium, for example, running overnight is nowadays considered as a viable option for many engineering design problems. The availability of cheap computer resources predetermines the need of automated search tools that can explore a huge design space in some (self) organized fashion. This dissertation has investigated the development and application of such search tools.

We have designed two core adaptive search engines, namely the ant colony and the inductive search. A major design criterion was that these search engines must be applicable to a variety of diverse problems ranging from well defined combinatorial optimization problems in chapter 2 to heavily constrained engineering design problems in chapter 6. Thus the definition of the core search tools was purposely kept quite generic.

Depending on the degree of *a priori* knowledge and complexity of the problem at hand some approaches turn out to be more efficient than others. For example, chapter 2 has investigated problems for which the search space can be readily made feasible through an appropriate selection of problem representation and operators. Whenever applicable, this approach has the advantage of concentrating search power into optimizing the main criteria rather than looking for feasible solutions. We have shown that for combinatorial optimization problems (COPs) the approach is readily applicable (sections 2.1 and 2.2). We have also shown that handling constraints by appropriate selection of problem representation is by no means only limited to COPs. There are many problems for which the feasible solutions can be mapped to local/global extrema of some related auxiliary cost function. This function can be viewed as a new problem representation. Section 2.3 has

shown how such an auxiliary cost function can be design for the protein-folding problem. The assumption is that the global minima of the energy landscape function are the admissible protein-folding configurations.

Handling constraints at the problem representation level utilizes a high degree of *a priori* knowledge and thus the design of search engines exploiting this approach is quite problem specific. Therefore, in chapter 3 we have shown how to explicitly derive the feasible region for a particular real world problem. Again, the advantage is that the feasible region can be effectively found before the search process begins. Results have clearly indicated that this approach significantly decreases the overall search time because most of the computational efforts are in optimizaing the main criterion.

However, it is not always possible to derive the feasible region in advance. Such problems need a search tool that uses both the constraint and the objective functions to guide the search process (chapter 4). This approach is less problem specific, but is less efficient, because it has to allocate computational resources for dealing with non-feasible solutions. This research area is relatively well developed mostly from a mathematical-programming point of view. Therefore, our main research efforts have been to view the ant colony search engine as a metaphor for combining existing techniques into coherent hybrid adaptive search systems. Results clearly indicated that when the "building blocks" are carefully chosen the implemented hybrid system is capable of efficiently achieving highly fit solutions.

In engineering design there are many problems (especially in the early design stages) for which the main goal is to find the feasible region. Chapter 5 has developed techniques to achieve this goal. Results have shown that the number of the cost function calls can be

significantly reduced while maintaining high quality of the results. Our approach has utilized some recent developments in numerical analysis and employed a natural analogy from the immune system.

When dealing with real world problems we often have to both outline the feasible region and optimize certain criteria. This becomes increasingly difficult, especially when the feasible region itself is hard to find. Chapter 6 has dealt with such a problem. It has shown how to combine various search approaches in order to find the most viable. Achieved results significantly outperformed the manual design procedure proving that the developed search techniques are an excellent decision support tool in the early stages of the design process.

In conclusion, the search tools developed in this dissertation proved to be efficient, robust and applicable to a diverse variety of design problems. They effectively utilize the available computing resources offered by advances in technology. Achieved results have shown that the search tools are capable of finding good solutions, many of which are novel to the engineering designer. As such, the developed tools are highly recommended to aid the decision support techniques in the preliminary stages of the engineering design process.

Appendix A

The Ant Colony Search Model

The main program looks like:

```
const float ACCURACY = 0.001;
int FE = 0;
#include "AntsLib.h"

main() {

        Ant NEST[ANTS_PATHS];
        int i;

// Initialize NEST

        for (i=0; i<ANTS_PATHS; i++) {
                for (int j=0; j<ARITY; j++)
                        NEST[i].x[j] = frandom(LOW[j], HIGH[j]);
                NEST[i].raw_f = f(NEST[i],0);
                for (j=0; j<CNSTR; j++)
                        NEST[i].cnstr[j] = constraint(NEST[i].x, j);
                NEST[i].df = 0;
                NEST[i].use_dir = 0;
                NEST[i].age = 0;
                NEST[i].id = i;
                NEST[i].trail = MIN_TRAIL;

        }

// MAIN LOOP

        for (int t=0; t<ANTS_GENERATIONS; t++) {

                qsort(NEST, ANTS_PATHS, sizeof(Ant), Ant_cmp_f);

                add_trail_f(NEST, ANTS_PATHS);

                diffuse_ants(NEST, ANTS_PATHS, t);

                random_walk(NEST, ANTS_PATHS - DIFFUSION, t);

                qsort(NEST, ANTS_PATHS, sizeof(Ant), Ant_cmp_tr);


                for (i=0; i<ANTS_NUMBER - DIFFUSION - RANDOM_WALK; i++) {
                        int index = wheel_selection(NEST, ANTS_PATHS);
                        go(NEST, index, t);

                }

                evaporate(NEST, ANTS_PATHS);
        }

// END MAIN LOOP

}
```

The search operators are located in the file **AntsLib.h**:

```
#include "Ants.h"

const  int          ANTS_GENERATIONS = 100;
const  int          ANTS_PATHS = 500;
const  int          ANTS_NUMBER = 100;
const  float        ANTS_NEIGHBOURHOOD = 0.02;
const  float        ANTS_N_EPS = 0.0000001;
const  int          DIFFUSION = 1;
const  int          RANDOM_WALK = 80;


const  int          DEATH_F = 37;
const  float        TRAIL_INC_F = 15.6448;
const  int          TRAIL_LEN_F = 19;


const  float  TRAIL_EVAPORATE = 100;

unsigned long seed = time(NULL);
ACG generator(seed, 98);


#include "bump.cc"  // Include the function to be solved here


float frandom(float l, float h) {
      float eps = 0.0000001;
      Uniform rnd(l-eps, h+eps, &generator);
      return rnd();
}

int irandom(int l, int h) {
      DiscreteUniform rnd(l, h, &generator);
      return (int)rnd();
}

int irandom(int l, int h, int prev) {
      DiscreteUniform rnd(l, h, &generator);
      int rand = (int)rnd();
      while ( rand == prev ) { rand = (int)rnd(); }
      return rand;
}


void display(const Ant *NEST, int N, ostream &out) {
      for (int i=0; i<N; i++)
            out << NEST[i] << endl;
}

float boundary(int t, int index) {
      float scale = HIGH[index] - LOW[index];
      if (t > DEATH_F) return (scale * ANTS_N_EPS);
      return (scale * (((ANTS_N_EPS - ANTS_NEIGHBOURHOOD) /
                              DEATH_F) * t + ANTS_NEIGHBOURHOOD));
}


float new_value(float val, int t, char dir, int index) {
      if (dir == 0) {
            float low = val - boundary(t, index);
            if (low < LOW[index]) low = LOW[index];
            float f = frandom(low, val);
            return f;
      } else {
            float high = val + boundary(t, index);
            if (high > HIGH[index]) high = HIGH[index];
            float f = frandom(val, high);
            return f;
      }
}
void go(Ant *NEST, int index, int t) {
      if ( (index > ANTS_PATHS) || (index < 0) ) {
            cerr << "go: range violation\n"; exit(1); }
```

```
        Ant new_ant;
        int new_dir[ARITY];

        for (int i=0; i<ARITY; i++) {
              if (NEST[index].use_dir != 0) new_dir[i] = NEST[index].dir[i];
              else new_dir[i] = irandom(0, 1);
              new_ant.x[i] = new_value(NEST[index].x[i],
                                              NEST[index].age, new_dir[i], i);
        }

        float new_f = f(new_ant, t);

        float cv[CNSTR];
        for (int p=0; p<CNSTR; p++) {
              cv[p] = constraint(new_ant.x, p);
        }


        if (new_ant.raw_f < NEST[index].raw_f) {
              for (i=0; i<ARITY; i++) {
                    NEST[index].x[i] = new_ant.x[i];
                    NEST[index].dir[i] = new_dir[i];
              }
              NEST[index].f = new_ant.f;
              NEST[index].raw_f = new_ant.raw_f;
              for (int k=0; k<CNSTR; k++)
                    NEST[index].cnstr[k] = cv[k];
              NEST[index].use_dir = 1;
              NEST[index].age = 0;
        } else {
              NEST[index].df = 0;
              NEST[index].age++;
              NEST[index].use_dir = 0;;
        }
}

int wheel_selection(const Ant *NEST, int N) {
        float sum = 0;
        for (int i=0; i<N; i++) sum += NEST[i].trail;

        float rand = frandom(0, sum);
        float tmp = NEST[0].trail;
        for (i=1; i<=N; tmp+=NEST[i].trail, i++)
              if (tmp >= rand) return (i-1);
        return (i-2);
}


void add_trail_f(Ant* NEST, int N) {
        float s = TRAIL_INC_F * TRAIL_LEN_F;

        int i=0, tr=0;
        while ( (tr < TRAIL_LEN_F) && (i < N) ) {
              if (NEST[i].age < DEATH_F) {
                    NEST[i].trail += s;
                    s -= TRAIL_INC_F;
                    tr++;
              } else
                          NEST[i].trail = 0;
              i++;
        }
}

void evaporate(Ant* NEST, int N) {
        for (int i=0; i<N; i++) {
              if (NEST[i].trail > MIN_TRAIL) {
                    NEST[i].trail -= TRAIL_EVAPORATE;
                    if ( NEST[i].trail < MIN_TRAIL) NEST[i].trail = MIN_TRAIL;
              }
        }
}


int Ant_cmp_f(const void* i, const void* j) {
        float eps = 0.0000000001;
        if ( (*(Ant*)i).f < (*(Ant*)j).f ) return 1;
```

```
        if ( (*(Ant*)i).f > (*(Ant*)j).f ) return -1;
        float tmp = ( (*(Ant*)i).raw_f - (*(Ant*)j).raw_f );
        if ( (tmp < eps) && (tmp > -eps) ) return 0;
        else
                if (tmp > eps) return 1;
                else return -1;
}

int Ant_cmp_tr(const void* i, const void* j) {
        float eps = 0.000001;
        float tmp = ( (*(Ant*)i).trail - (*(Ant*)j).trail );
        if ( (tmp < eps) && (tmp > -eps) ) return 0;
        else
                if (tmp > eps) return -1;
                else return 1;
}


float calc_new_trail(Ant *NEST, int N, float raw_f, float feasibility) {
        float new_trail = MIN_TRAIL;

        for (int i=0; i<N; i++)
                if ( (NEST[i].raw_f > raw_f) && (NEST[i].f == feasibility) )
                        if (new_trail < NEST[i].trail)
                                new_trail = NEST[i].trail;
        return new_trail;
}


void diffuse_ants(Ant *NEST, int N, int t) {

        for (int ant=1; ant <= DIFFUSION; ant++) {
                Ant             new_ant;

                for (int i=0; i<ARITY; i++) {
                        int a1 = irandom(0, TRAIL_LEN_F);
                        int a2 = irandom(0, TRAIL_LEN_F);
                        while (a1 == a2) {
                                a2 = irandom(0, TRAIL_LEN_F);
                        }
                        int alt = irandom(0, 2);
                        switch (alt) {
                                case 0: new_ant.x[i] = NEST[a1].x[i]; break;
                                case 1: new_ant.x[i] = NEST[a2].x[i]; break;
                                case 2: float b = frandom(0.1, 0.9);
                                        new_ant.x[i] = (NEST[a1].x[i] * b  +
                                                        NEST[a2].x[i] * (1-b));
                        }
                }



                NEST[N-ant].age = 0;
                NEST[N-ant].use_dir = 0;
                NEST[N-ant].df = 0;
                NEST[N-ant].raw_f = f(new_ant,t);;
                NEST[N-ant].f = new_ant.f;
                NEST[N-ant].trail = calc_new_trail(NEST, N,
                                        NEST[N-ant].raw_f, NEST[N-ant].f);
                for(i=0; i<ARITY; i++)
                        NEST[N-ant].x[i] = new_ant.x[i];
                for (i=0; i<CNSTR; i++)
                        NEST[N-ant].cnstr[i]=constraint(NEST[N-ant].x,i);
                NEST[N-ant].id = -NEST[N-ant].id;
        }
}


float delta(float rel_time, float range, float r) {
        int b = 10;
        const float eps = 0.00001;
        if (rel_time >= 1 - eps) return eps;
        else return ( range * ( 1 - pow( r,
                        pow( (double)(1 - rel_time), (long)b))));
}
```

```
float dynamic_mutation(float f, float low, float high, int t) {
        float range;
        float r = frandom(0,1);
        if (irandom(0,1) == 1) {
                range = high - f;
                float val = f + delta(t/(double)ANTS_GENERATIONS,range, r);
                return val;
        } else {
                range = f - low;
                float val = f - delta(t/(double)ANTS_GENERATIONS, range, r);
                return val;
        }
}


void random_walk(Ant *NEST, int N, int t) {
        for (int ant = 1; ant <= RANDOM_WALK; ant++) {
                Ant    new_ant;

                for (int i=0; i<ARITY; i++) {
                        int a = irandom (0, TRAIL_LEN_F);
                        if (irandom(0, 1) == 1)
                                new_ant.x[i] = dynamic_mutation(NEST[a].x[i],
                                                                LOW[i], HIGH[i], t);
                        else
                                new_ant.x[i] = NEST[a].x[i];
                }

                NEST[N-ant].raw_f =f(new_ant, t);;


                NEST[N-ant].age = 0;

                NEST[N-ant].use_dir = 0;
                NEST[N-ant].df = 0;

                NEST[N-ant].f = new_ant.f;
                NEST[N-ant].trail = calc_new_trail(NEST, N,
                                                   NEST[N-ant].raw_f, NEST[N-ant].f);
                for (i=0; i<ARITY; i++)
                        NEST[N-ant].x[i] = new_ant.x[i];
                for (i=0; i<CNSTR; i++)
                        NEST[N-ant].cnstr[i] = constraint(NEST[N-ant].x,i);
        }
}
```

The basic data structures are defined in **Ants.h**:

```
#include <ACG.h>
#include <Uniform.h>
#include <DiscUnif.h>
#include <Normal.h>
#include <sys/time.h>
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>

const float MIN_TRAIL = 1;
const int    MAX_ARITY = 250;
const int MAX_CNSTR = 50;


struct Ant {

//      State of the Search

        float  x[MAX_ARITY];
        float cnstr[MAX_CNSTR];
        float  f;
        float  raw_f;
        float  df;
        int          dir[MAX_ARITY];
        int          use_dir;
```

```
//      State of the Path

        float  trail;
        int          age;
        int          id;

//      Constructor

        Ant()  {trail = MIN_TRAIL; age=0; df=0; use_dir=0; id = -1; f=0;
                  for (int i=0; i<MAX_ARITY; i++) { x[i] = 0; dir[i] = 0; } }
};

ostream& operator << (ostream& out, const Ant& a) {
        out
                << "id: " << setw(4) <<a.id
                << "  age: " << setw(5) << a.age
                << "  tr: " << setw(5) << a.trail
                << " $$$ r: " << setw(10) << setprecision(8) << a.raw_f
                << " *** f: " << setw(12) << setprecision(10) << a.f
                << setprecision(5)
                << "  | " << a.cnstr[0]
                << "    " << a.cnstr[1];

        return out;
}
```

The problem to be solved is included in the following format:

```
const int ARITY = 13;
const float LOW[ARITY] = { 0,0,0,0,0,0,0,0,0,  0,   0,   0, 0};
const float HIGH[ARITY] = {1,1,1,1,1,1,1,1,1,100, 100, 100, 1};
const int CNSTR = 9;

float constraint(float *x, int i) {
        float eps = 0.00000000001;
        switch (i) {
                case 0: return -2*x[0]-2*x[1]-x[9]-x[10]+10;
                case 1: return -2*x[0]-2*x[2]-x[9]-x[11]+10;
                case 2: return -2*x[1]-2*x[2]-x[10]-x[11]+10;
                case 3: return 8*x[0]-x[9];
                case 4: return 8*x[1]-x[10];
                case 5: return 8*x[2]-x[11];
                case 6: return 2*x[3]+x[5]-x[9];
                case 7: return 2*x[5]+x[6]-x[10];
                case 8: return 2*x[7]+x[8]-x[11];
                default: cerr << "Unknown constraint number\n"; exit(1);
        }


}

float tc(float *x) {
        float tmp = 0;
        float tmp2 = 0;
        for (int i=0; i<4; i++) tmp += x[i]*x[i];
        for (i=4; i<ARITY; i++) tmp2 += x[i];

        return 5*x[0]+5*x[1]+5*x[2]+5*x[3]-5*tmp-tmp2;
}


float DROP = -1000000;
float Z = 0.7;
float HIGH_OPTIMISM = ACCURACY;
float LOW_OPTIMISM =  ACCURACY;
float STOP_REWARDING_OPTIMISM =  Z * ANTS_GENERATIONS;

float OPTIMISM(int t) {
        if (t > STOP_REWARDING_OPTIMISM) return LOW_OPTIMISM;
        return  ((LOW_OPTIMISM - HIGH_OPTIMISM) /
                                STOP_REWARDING_OPTIMISM ) * t + HIGH_OPTIMISM;

}
```

```
float f(float *x, int t) {
        FE++;
        float cv = 0;
        float tmp;
        for (int i=0; i<CNSTR; i++) {
                tmp = constraint(x,i);
                if ( tmp  < 0) cv += fabs(tmp);
        }

        if (cv <= OPTIMISM(t)) return DROP +  tc(x);

        return cv;

}
```

Appendix B

The Inductive Search Model

The main program looks as follows:

```
#include <Normal.h>
#include <Uniform.h>
#include <DiscUnif.h>
#include <ACG.h>
#include <sys/time.h>
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <LEDA/sortseq.h>
#include <LEDA/stream.h>
#include <math.h>
#include <iomanip.h>
#include "main.h"

/*
**      Include a test case here:
**
*/

#include "t1.c"


float func(float y)

/*
**
**      Auxiliary 1-D function
*/

{
        iter_count++;

        x[DIM-1] = y;
        float res =  f(x,DIM);
        return res;
}


float func_learning(float *y)

/*
**
**      Auxiliary DIM-dimensional function
*/

{
        iter_count++;
        float res = f(y,DIM);
        return res;
}


void oracle(int dim)

/*
**      One possible deterministic implementation
**        of the non-deterministic ORACLE function
*/

{
        float r,bren,xmin;

        float ax = LOW;
        float cx = HIGH;
        float bx = (ax+cx)/2;
```

```
        DIM = dim;

        sortseq<float,Interval> Population;
        seq_item S;
        Interval I;
        I.low = ax;
        I.high = cx;
        Population.insert(cx-ax,I);

        int STOP = 0;
        int count = 0;
        float FMIN = 1e30, XMIN;

        while (STOP == 0) {

                S = Population.max();
                Population.del_item(S);
                I = Population.inf(S);

                ax = I.low;
                cx = I.high;
                bx = (ax+cx)/2;

                bren = brent(ax,bx,cx,func,TOL,&xmin);


                if (bren < FMIN) { FMIN = bren; XMIN = xmin; }


                if (bx < xmin) {
                        I.low = ax; I.high = bx; Population.insert(I.high-I.low, I);
                        I.low = bx; I.high = xmin;
                        Population.insert(I.high-I.low, I);
                        I.low = xmin; I.high = cx;
                        Population.insert(I.high-I.low, I);
                } else {
                        I.low = ax; I.high = xmin;
                        Population.insert(I.high-I.low, I);
                        I.low = xmin; I.high = bx;
                        Population.insert(I.high-I.low, I);
                        I.low = bx; I.high = cx; Population.insert(I.high-I.low, I);
                }

        count++;
        if (count > DELTA_N) break;
        }


//      Local learning

        x[DIM-1] = XMIN;

        if ( (LEARNING) && (DIM > 1) ) {
                local_learn(func_learning, x, DIM);
        }
}


main()
{
        for(DELTA_N=STOP_CRIT; DELTA_N<=STOP_CRIT; DELTA_N++) {
                iter_count = 0;
                for (int i=0; i<NDIM; i++) {
                        oracle(i+1);
                        cout << "     dim: " << setw(3) << i+1;
                        cout << "     f: " << setw(7) << f(x,i+1);
                        cout << endl;
                }
        }
}
```

The help routines are located in **main.h**:

```
extern "C" float brent(float ax, float bx, float cx,
                       float (*f)(float), float tol, float *xmin);
#define TOL 1.0e-4

unsigned long seed = time(NULL);
ACG generator(seed, 1000);

#include "rand.c"

#define NDIM 10

float  x[NDIM];
int    DIM=0;
int DELTA_N;

static int iter_count = 0;

#include "dhc.c"            // Local hill climber


struct Interval {
      float low,high;
};

void Print(Interval &I, ostream& out) {
      out << I.low << "    " << I.high;
}

void Read(Interval &I, istream& in) {
      in >> I.low >>  I.high;
}

void print(sortseq<float,Interval>& S)
{ seq_item it;
  forall_items(it,S) cout << S.key(it) << " ==> " << S.inf(it).low
                      << "    " << S.inf(it).high << endl;
  newline;
}
```

The problem to be solved is given in ythe following format:

```
#define STOP_CRIT 0
#define LEARNING 0

#define LOW -5.0
#define HIGH 5.0


float f(float *x,int n)
{

      register int i;
      float        Sum;

      for (Sum = 0.0, i = 0; i < n; i++) {
            Sum += x[i]*x[i];
      }

      return (Sum);
}
```

Appendix C

Method of Hooke and Jeeves

The algorithm of Hooke and Jeeves with improvements due to Bell and Pike [Bell and Pike, 1969], and Smith [Smith, 1969] is defined as follows:

Step 0: (***Initialization***)
Choose a starting point $x^{(0,0)}$, an accuracy bound $\varepsilon > 0$, and initial step lengths $s_i^{(0)} \neq 0$ for all $i = 1$ to n (e.g. $s_i^{(0)} = 1$ if no more plausible values are at hand).
Set $k = 0$ and $i = 1$.

Step 1: (***Exploratory move***)

Construct $x' = x^{(k,\, i-1)} + s_i^{(k)}\, e_i$	(***discrete step in positive direction***);
If $F(x') < F(x^{(k,\, i-1)})$, go to step 2	(***successful first trial***);
otherwise replace $x' \leftarrow x' - 2\, s_i^{(k)}\, e_i$	(***discrete step in negative direction***);
If $F(x') < F(x^{(k,\, i-1)})$, go to step 2	(***success***);
otherwise replace $x' \leftarrow x' + s_i^{(k)}\, e_i$	(***back to original situation***).

Step 2: (***Retension and switch to next coordinate***)
Set $x^{(k,\, i)} = x'$.
If $i < n$, increase $i \leftarrow i + 1$ and go to step 1.

Step 3: (***Test for total failure in all directions***)
If $F(x^{(k,\, i)}) \geq F(x^{(k,\, 0)})$, set $x^{(k+1,\, 0)} = x^{(k,\, 0)}$ and go to step 9.

Step 4: (***Pattern move***)

Set $x^{(k+1,\, 0)} = 2\, x^{(k,\, n)} - x^{(k-1,\, n)}$	(***extrapolation***);
and $s_i^{(k+1)} = s_i^{(k)}\, \mathrm{sign}(x_i^{(k,\, n)} - x_i^{(k-1,\, n)})$	(***this may change the sequence of positive and negative directions in the next exploratory move***);
Increase $k \leftarrow k + 1$ and set $i = 1$.	(***note: there is no success control of the pattern move so far***).

Step 5: (***Exploration after extrapolation***)
Construct $x' = x^{(k,\, i-1)} + s_i^{(k)}\, e_i$
If $F(x') < F(x^{(k,\, i-1)})$, go to step 6;
otherwise replace $x' \leftarrow x' - 2\, s_i^{(k)}\, e_i$
If $F(x') < F(x^{(k,\, i-1)})$, go to step 6;
otherwise replace $x' \leftarrow x' + s_i^{(k)}\, e_i$

Step 6: (***Inner loop over coordinates***)
Set $x^{(k,\,i)} = x'$.
If $i < n$, increase $i \leftarrow i + 1$ and go to step 5.

Step 7: (***Test for fialure of pattern move***)
If $F(x^{(k,\,n)}) \geq F(x^{(k-1,\,n)})$ (***back to position before pattern move***);
set $x^{(k+1,\,0)} = x^{(k-1,\,n)}$, $s_i^{(k+1)} = s_i^{(k)}$
for all $i = 1$ to n, and go to step 10.

Step 8: (***After successful pattern move, retension and first termination test***)
If $\frac{1}{2} | s_i^{(k)} | \geq | x_i^{(k,\,n)} - x_i^{(k-1,\,n)} |$ for all $i = 1$ to n,
set $x^{(k+1,\,0)} = x^{(k-1,\,n)}$ and go to step 9;
otherwise go to step 4 for another pattern move.

Step 9: (***Step size reduction and termination test***)
If $\varepsilon \geq | s_i^{(k)} |$ for all $i = 1$ to n end the search with result $x^{(k,\,0)}$;
otherwise set $s_i^{(k+1)} = \frac{1}{2}\, s_i^{(k)}$ for all $i = 1$ to n.

Step 10: (***Iteration loop***)
Increase k $\leftarrow k + 1$, set $i = 1$, and go to step 1.

Appendix D

Protein-folding Free Energy Model:

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <math.h>

#define NDIM 3

int P[NDIM+2];

static double       x[NDIM];


double C(int a1, int a2) {
      return 0.125*(1+a1+a2+5*a1*a2);
}

double V1(double x) {
      return 0.25*(1-cos(x));
}


double V2(double r, int a1, int a2) {
      double r12 = pow(r, -12);
      double r6 = pow(r,-6);
      return 4*(r12-C(a1,a2)*r6);
}


double r(int i, int j, double *x) {
      double cos_sum=0;
      double sin_sum=0;
      double y[NDIM+1]; y[0]=0;
      for (int z=0; z<NDIM; z++) y[z+1] = x[z];

      for (int k=i; k<=j-1; k++) {
            double sum = 0;
            for (int l=i; l<=k; l++) {
                  sum += y[l];
            }
            cos_sum += cos(sum);
            sin_sum += sin(sum);
      }
      return sqrt(cos_sum*cos_sum + sin_sum*sin_sum);
}


double f(double *x, int n) {
      n = n+2;
      double sum1 = 0;
      int i;
      for (i=0; i<n-2; i++) {
            sum1 += V1(x[i]);
      }

      double sum2 = 0;
      for (i=0; i<n-2; i++)
            for(int j=i+2; j<=n-1;j++) {
                  sum2 += V2(r(i,j,x),P[i],P[j]);
            }
      return sum1 + sum2;
}
```

Appendix E

The Fault Population Used in Chapter 3.

Each entry represents a possible fault and lists the pattern that discovers it:

```
**1*00*1***1
*1***1010010
1*01***00**0
1***0***1***
*01*11****00
*010*0*0*10*
1*0*0*110***
110*1111*0**
10*11101***0
**0****1*011
*0**0*1*****
000*00***1**
1*1***11*1*0
1****1*0*111
**1*1*11**00
1**00*0*1101
0*000010001*
**01*11101**
11****0110**
11**1*10*01*
0010100*1*0*
**0*1**10***
0***00*111*1
1*1******1*0
*****00*0101
0**1***1*010
11*0*0**1*01
0*011**0*010
11*11***001*
0*****1**01*
*0110**0**01
00*1100****0
**0**01*1**0
11**01**1***
***1*010110*
*1*010**1**1
1*010*1*100*
*0**101**10*
*0***0001**1
******00*0**
*0*11**01***
00***10****1
*0110**1**11
1*1**0****1*
*0**0*101*1*
11**1*1*****
*0*11***0*1*
0**00***01*0
**0**00***11
***10110****
10***0**1*11
*****01100*0
***100****1*
00001*****10
0*01***10*0*
110**111111*
0**1*0*0**01
**1***010*01
*1**10*0*010
*01*0*0*01**
```

***1**0*1**0
*1*1010*1*01
1**10*0****1
00**0*1*0**0
*01**1**1***
1*1000**0**1
1*100**0*1**
1**1*00010*0
00**11**101*
0**1*011*0**
***0000**1**
*1*1**00**00
1****00*0*1*
01*01***
10010**0***0
1111****0**0
*11***0**010
*01**1*01101
1011*010*0**
10***1*000**
111*0****000
1000*0*0*10*
****01**0011
10*00***1***
*00**0**11**
1*1**01*1010
1110**1110**
1010*0**10*1
*0*1*01*****
1*0**0*1*1*1
11*0**101**1
****0*1***1*
110*0**0
11*00*1*1***
*01*0**100**
1**110***010
00*0***0*0*1
001*01*1*0*0
00*010*1
10***1****0*
*01***010***
0*001100
**1011101100
10**11011*10
1*0*1*0**11*
1*1*01111***
*0*01*1**0*0
*1110*010110
110*00*010*0
11**001*1011
1100*0*1*10*
*1**0*0*0110
*011*10*****
1*000*00011*
1****0*00**1
*1*010*10**1
10****00***1
1011*1*0
00******1*10
1*0***
1***011*000*
1*0**00100**
*01*1*0**100
10*****1**0*
*00***10**1*
110***00
01*0*1
1*10*0*0*1**
*0*1*0**101*
**01*1*01000
11110101

*****101****
*0111**1*01*
0*11*01*110*
0*11****00*1
1*0**00**0*1
0*0**0011000
*1***000*1**
00*10*0*110*
****00**1***
***001*11**1
00*1**0****1
111*0*11
***00**1*0*0
1**1******01
10*0**001*01
*01*0***1010
0001*****00*
1110*1******
00*0**00****
001010
***010*0*11*
1***10*1**10
00*0*1011**0
00011**101**
0******00*01
***0*0*0**10
****0****100
**1000010*1*
1*******00*0
11**11
100**1*01*11
11****0*010*
*101**0**0**
*0**1****000
1*0*11*****0
1*0**01*001*
0****0110*0*
*****1***00*
***110*000**
*01*111**101
*11*1*1*0*0*
*****1*0*1**
110*111****0
11****11**0*
*0**01011**0
0**00***1***
1*****10110*
10*1*000
*1101*011*0*
11100**01*1*
*000*1010***
1**0**10**0*
*0010*001*1*
1010**0001*0
****01***1*0
*01000****00
0**0*1110*1*
11**1*0*011*
1*11**001**1
10*0***
1**1*010*10*
*0101*11****
*1*11*1**1*0
00***0*101*0
1**101101011
1*00*0110011
*01*1*****10
0***0**0****
**1001*0*11*

Appendix F

Test Cases used in Chapter 5.

```
// Used by all testbeds
double f_base(double x, double y, double a, double b, double c, double d)
{
       return exp(-(a*x-b)*(a*x-b)-(c*y-d)*(c*y-d));
}
//

//******************
//Testbed Non-convex
//******************

double f1(double x, double y)
{
       return f_base(x,y,0.4,0,0.4,-0.5)+f_base(x,y,0.4,-1.5,0.4,-0.5);
}

double f2(double x, double y)
{
       return f_base(x,y,0.4,0,0.4,0.5)+f_base(x,y,0.4,-2.5,0.4,0.5);
}


double f(double x, double y)
{
       double f11=f1(x,y), f22=f2(x,y);
       double cv1=0, cv2=0;
       if (f11 < 0.41) cv1 = 0.41 - f11;
       if (f22 < 0.41) cv2 = 0.41 - f22;
       return cv1+cv2;
}


//*****************************************************
//Testbed 2: Equal sizes of the feasible subregions;
//*****************************************************

double f1(double x, double y)
{
       return
f_base(x,y,0.4,2.5,0.4,0.5)+f_base(x,y,0.4,0,0.4,0.5)+f_base(x,y,0.4,-
2.5,0.4,0.5);
}

double f2(double x, double y)
{
       return f_base(x,y,0.4,2.5,0.4,-0.5)+f_base(x,y,0.4,0,0.4,-
0.5)+f_base(x,y,0.4,-2.5,0.4,-0.5);
}


double f(double x, double y)
{
       double f11=f1(x,y), f22=f2(x,y);
       double cv1=0, cv2=0;
       if (f11 < 0.7) cv1 = 0.7 - f11;
       if (f22 < 0.7) cv2 = 0.7 - f22;

       return cv1+cv2;
}

//*****************************************************
//Testbed 3: Different sizes of the feasible subregions;
//*****************************************************

double f1(double x, double y)
{
       return
f_base(x,y,0.4,2.7,0.4,0.4)+f_base(x,y,0.4,0,0.4,0.5)+f_base(x,y,0.4,-
2.5,0.4,0.5);
}
```

```
double f2(double x, double y)
{
	return f_base(x,y,0.4,1.5,0.4,-0.5)+f_base(x,y,0.4,0,0.4,-
0.5)+f_base(x,y,0.4,-1.5,0.4,-0.5);
}
double f(double x, double y)
{
	double f11=f1(x,y), f22=f2(x,y);
	double cv1=0, cv2=0;
	if (f11 < 0.56) cv1 = 0.56 - f11;
	if (f22 < 0.56) cv2 = 0.56 - f22;
	return cv1+cv2;
}
```

References

Antonov I.A., and V.M. Saleev (1979). *USSR Computational Mathematics and Mathematical Physics*, vol. 19, no. 1, pp. 252-256.

Bäck T. (1992). Self-Adaptation in Genetic Algorithms, in Varela F. and Bourgine P (eds.), *Procs. of the First European Conference on Artificial Life*, Cambridge, MA, MIT Press, pp. 263-271.

Baker B.S. (1985). A new proof for the first fit decreasing bin-packing algorithm, *J. Algorithms* **6**, pp. 49-70.

Bell M., and M.C. Pike (1969). Remark on algorithm 178 (E4) - direct search, CACM **9**, 684-685.

Bilchev G. (1994). Evolutionary Algorithms for the Bin Packing Problem, *MSc Thesis*, New Bulgarian University, (in Bulgarian).

Bilchev G, and I. Parmee (1995a). Adaptive Search Strategies for Heavily Constrained Design Spaces, in *Procs. of the 22nd International Conference on CAD-95*, Ukraine, Yalta, May 8-13.

Bilchev G. and I. Parmee (1995b). The Ant Colony Metaphor for searching Continuous Design Spaces, in *LNCS 993: Evolutionary Computing 2*, edited by T. Fogarty, Springer-Verlag, pp. 25-39.

Bilchev G. (1996). Evolutionary Metaphors for the Bin Packing Problem, *5th Annual Conference on Evolutionary Programming*, Feb 29-Mar 2, San Diego, USA, pp. 333-341.

Bilchev G. and I. Parmee (1996a). Constrained Optimisation with an Ant Colony Search Model, *Adaptive Computing in Engineering Design and Control '96*, March '96, University of Plymouth, UK, pp. 145-151.

Bilchev G., and I. Parmee (1996b). "Inductive Search", *First International Contest on Evolutionary Optimization , 1996 IEEE Conference on Evolutionary Computation*, May 20-22, Nagoya, Japan, pp. 832-836[1]

Bilchev G., and I. Parmee (1996c). "Learning the 'Next' Dimension", *Artificial Intelligence and Simulation of Behaviour'96*, Brighton, UK, April '96, pp. 162-174

Bilchev G., and I. Parmee (1996d). The Inductive Genetic Algorithm with Applications to the Fault Coverage Test Code Generation Problem, *EUFIT'96*, Aachen, Germany, September '96.

Bilchev G., and I. Parmee (1996e). Constrained Handling for the Fault Coverage Code Generation Problem: An Inductive Evolutionary Approach, PPSN IV, Sept. 96, Berlin, *LNCS 1141*, Springer, pp. 880-889.

Bledsoe W. W. (1985). The use of biological concepts in the analytical study of systems, presented at the ORSA-TIMS National meeting, San Fransisco, CA

Bouchard E., *et. al.* (1988). The application of artificial intelligence technology to aeronautical system design, AIAA-88-4426, AIAA/AHS/ASEE Aircraft Design Systems and Operations Meeting, Septmber 7-9, Atlanta.

Bratley P. and B.L. Fox (1988). *ACM Trans. on Math. Software*, vol. 14, pp. 88-100.

Cea J. (1971). Optimisation - theorie et algorithmes, Dunod, Paris.

Clearwater S., Huberman B., and Hogg T. (1992). Cooperative Problem Solving, in B. Huberman, editor, *Computation: The Micro and the Macro View*, pp. 33-70, World Scientific, Singapore.

Colorni A., Dorigo M., and Maniezzo V. (1991). Distributed Optimization by Ant Colonies, in *Procs. First European Conference on Artificial Life*, Varela F., and Bourgine P. (eds.), Paris, Elsevier, pp. 134-142.

Cook S. A. (1971). The complexity of theorem proving procedures, in *Procs. of the Third Annual ACM Symposium on Theory of Computing*, ACM.

Creighton T (1984). Proteins, Structures and Molecular Principles, Freeman, New York.

Dixon J. (1986). Artificial Intelligence and Design, A Mechanical Engineering View, *Procs. of AAAI.*

Dorigo M. (1992). Optimization, Learning, and Natural Algorithms, *PhD Thesis*, Politecnico di Milano, ITALY, (in Italian).

Falkenauer E. (1994). New Representation and Operators for Genetic Algorithms Applied to Grouping Problems, in *Evolutionary Computation*, Vol.2 , No. 2, pp. 123-144.

Falkenhainer B., and K. Forbus (1988). Compositional modeling: finding the right model for the job, *Artificial Intelligence* **51**(1-3), 95-143.

Feigenbaum E., J. Feldman, ed. (1963). "Computers and Thought", McGraw-Hill, New York.

First International Contest on Evolutionary Optimization,
http://iridia.ulb.ac.be/langerman/ICEO.html

Fletcher R. (1987). Practical Methods of Optimization, Second edition, Wiley, Chichester.

Fogel L., A. Owens, and M. Walsh (1966). Artificial Intelligence through Simulated Evolution, New York, Wiley.

Fogel L., Angeline P., and Fogel D. (1995). An Evolutionary Programming Approach to Self-Adaptation on Finite State Machines, in Evolutionary Programming IV: *Procs. of the Fourth Annual Conference on Evolutionary Programming*, Cambridge, MA, The MIT Press.

Forrest S., B. Javornik, R. Smith, and A. Perelson (1993). Using Genetic Algorithms to Explore Pattern Recognition in the Immune System, *Evolutionary Computation*, Volume 1, Number 3.

Fujiwara H. and T. Shimono (1983). On the accelaration of test generation algorithms, *IEEE Transactions on Computers*, C-31:1137-1144.

Fujiwara H., and S Toida (1982). The complexity of fault detection problems for combinational logic circuits, *IEEE Transactions on Computers*, C-30:555-560.

Gaede R. K., *et. al.* (1986). CATAPULT: Concurrent Automatic Testing Allowing Parallelization and Using LimitedTopology, *Procs. of the 25th Design Automation Conference*, June 86.

Garey M. and Johnson D. (1979). Computers and Intractability - A Guide to the Theory of NP-completeness, W.H.Freeman Co., San Francisco, USA.

Gero J. (1992). Design prototypes: a knowldge representation schema for design, *AI Magazine* **11**(4), 27-36.

Gierasch L, and J. King (eds.) (1990). Protein Folding, AAAS, Washington.

Gill P., W. Murray and M Wright (1981). "Practical Optimization", Academic Press.

Goel P. (1981). An implicit enumeration algorithm to Generate Tests for Combinational Circuits, *IEEE Transactions on Computers*, C-30, No. 3, March '81, pp. 215-222.

Goldberg D. E. (1989). Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, Reading, MA.

Gross M., Ervin S., Anderson J., and Fleisher A. (1987). "Designing with Constraints", in *Computability of Design*, Y. E. Kalay ed., New Yaork: Wiley.

Halton J. H. (1960). Numerische Mathematik, vol. 2, pp. 84-90.

Holland J. H. (1975). Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications in Biology, Control, and Artificial Intelligence, University of Michigan Press, Ann Arbor.

Hooke, R., T.A. Jeeves (1961). Direct search solution of numerical and statistical problems, JACM **8**, 212-229.

Ibarra O. H., and S. K. Sahni (1975). Polynomially complete fault detection problems, *IEEE Transactions on Computers*, C-24:242-249.

Keane A. (1994). Experiences with optimizers in structural deign, *Procs. of the !st International Conference in Adaptive Computing in Engineering Design and Control*, University of Plymouth, UK, 1994, pp. 14-27.

Kinzel W. (1985). Learning and Pattern Recognition in Spin Glass Models.

Kirkpatrick S., Gelatt C.D., and Vechi M. P. (1983). Optimisation by Simulated Annealing, *Science*, Vol. 220, No. 4598, May '83.

Kowalik J. and M. R. Osborne (1968). Methods for Unconstrained Optimisation Problems, Elsevier, New York.

Koza J. (1992). Genetic Programming, Cambridge MA, MIT Press.

Lawrence C., J. Zhou, and A. Tits (1996). User's Guide for CFSQP Version 2.2: A C Code for Solving (Large Scale) Constrained Non-linear (Minimax) Optimization Problems, Generating Iterates Satisfying All Inequality Constraints, Electrical Engineering Dept. and Inst. for Systems Research, Univ. of Maryland, College Park, MD 20742.

Maher M.L. (1989). "Synthesis and evaluation of preliminary designs", in *Artificial Intelligence in Design*, J.S. Gero ed., New York: Springer-Verlag.

Martello S. and Toth P. (1990). Bin Packing Problem, Chapter 8 in Knapsack Problems, Algorithms and Computer implementations, John Wiley and Sons Ltd., England.

Michalewicz Z. (1992). Genetic Algorithms + data Structures = Evolutionary Programs, Springer-Verlag.

Michalewicz Z. (1995a). A Survey of Constraint Handling Techniques in Evolutionary Computation Methods, *The 4th Annual Conference on Evolutionary Programming' 95*, March 1-3, San Diego, USA.

Michalewicz Z. (1995b). Genetic Algorithms, Numerical Optimization, and Constraints, *Fourth Intl. Conference on Evolutionary Programming*, March 1-3, San Diego, USA.

Michalewicz Z., Nazhiyath, and Michalewicz M (1996). A Note on Usefulness of Geometrical Crossover for Numerical Optimization problems, *the 5th Annual Conference on Evolutionary Programming' 96*, San Diego, USA.

Nall B, and K. Dill (1991). Conformations and Forces in Protein Folding, AAAS, Washington.

Niederreiter H, P. Bratley, and B. L. Fox (1994). Algorithm 738: Programs to generate Niederreiter's low-discrepancy sequences, *ACM Trans. on Math. Software*, 20, 494-495.

Panier E.R., and A.L. Tits (1993). On combining feasibility, descent and superlinear convergence in inequality constrained optimization, *Math. Programming* 59, 261-276.

Parmee I., (1996). Cluster-Oriented Genetic Algorithms (COGAs) for the Identification of High Performance Regions of Design Spaces, EvCA96, Moscow, June 24-27.

Parmee I. (1995a). High-level Decision Support for Engineering Design Using the Genetic Algorithm and Complementary Techniques, *Procs. Applied Decision Technologies, Stream 2 "Modern Heuristic Search Methods"*, Brunel Conference Centre, Unicom, London, 3-5 April '95.

Parmee I. (1995b). Reinforcing the natural clustering tendencies of the genetic algorithm, Internal report PEDC-04-95, University of Plymouth, UK.

Parmee I., Johnson M, and Burt S. (1994). Techniques to Aid Global Search in Engineering Dign, *Procs. of International Conference on Industrial and Engineering Applications of AI and Expert Systems*, Austin, Texas.

Parmee I. and M.J. Denham (1994). Emergent Computing Methods in Engineering Design, NATO Advanced Research Workshop, Nafplio, Greece, August '94.

Powell D., and M Skolnick (1993). Using Genetic algorithms in engineering design optimization with non-linear constraints, *Procs. of the 5th International Conference on Genetic Algorithms*, University of Illinois at Urbana-Champaign, pp 424-431.

Press W., Teukolsky S., Vetterling W., and Flannery B (1992). Numerical Recipes in C, Cambridge Univ. Press, p. 402.

Radcliffe N. (1991). Forma Analysis and Random Respectful Recombination, in *Procs. of the Fourth ICGA, San Diego.*

Radcliffe N. and Surry P. (1995). Fundamental Limitation Theorems on Search Algorithms: Evolutionary Computing in Perspective, *LNCS 1000*, Springer-Verlag.

Ratschek H., J. Rokne (1988). "New Computer Methods for Global Optimization", Ellis Horwood ltd.

Richardson J, and M Palmer (1989). Some guidelines gor Genetic algorithms with penalty functions, *Procs. of the 3rd International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Los Atlos, CA, pp. 191-197.

Ross D. E., M. R. Mercer (1990). WAVE, A Concurrent Approach to Combinational Test Pattern Generation, *Procs. of the MCC University Research Symposium.*

Roth J. P. (1966). Diagnosis of Automata Failure: A Calculus and a Method, IBM Journal of Research and Development, Vol, 10, July '66, pp. 278-291.

Rumelhart D.E., Hinton G.E., and Williams R.J. (1986). Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1, The MIT Press, Cambridge, MA.

Schoenauer M, and S. Xanthakis (1993). Constrained GA optimization, *Procs. of the 5th International Conference on Genetic Algorithms*, University of Illinois at Urbana-Champaign, pp 573-580.

Schulz M. H. *et. al.* (1988). SOCRATES: A highly efficient automatic test pattern generation system, *IEEE Transactions on CAD*, pp. 126-137, January '88.

Smith L.B., F.K. Tomlin (1969). Remark on algorithm 178 (E4) - direct search, CACM **12**, 637-638.

Smith R., S. Forrest, and A. Perelson (1993). Population Diversity in an Immune System Model: Implications for Genetic Search, *Procs. of the 5th International Conference on Genetic Algorithms*, University of Illinois at Urbana-Champaign.

Sobol I. M. (1967), *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86-112.

Some Hard Global Optimization Test Problems,
http://solon.cma.univie.ac.at/~neum/glopt/my_problems.html

Stanion T., D. Bhattacharya (1991). TSUNAMI: A Path Oriented Scheme for Algebraic Test Generation, *Procs. of Fault Tolerant Computing Symposium*, June '91, pp. 36-43

Steels L. (1988). Artificial Intelligence and Complex Systems, AI-MEMO 88-2, AI-lab, VUB, Brussels.

Stillinger F, T. Head-Gordon, and C. Hirshfeld (1993). Toy model for protein forlding, Physical Review E, **48**(2), pp 1469-1477.

Törn A., and Zilinskas A. (1988). Global Optimization, *Lecture Notes in Computer Science 350*, Springer-Verlag.

Traub J. (1996). On Reality and Models, Santa Fe Institute Technical Report 96-03-010.

Weisbuch G. (1991). Complex Systems Dynamics, Lecture Notes Volume II, Santa Fe Institute, Studies in the Sciences of Complexity, Addison-Wesley.

Wolpert D. and Macready W. (1995). No Free Lunch Theorems for Search, Santa Fe Institute, SFI-TR-95-02-010.

Yuret D. (1994). From Genetic Algorithms to Efficient Optimization, *MSc thesis*, MIT May '94.

Zienkiewicz O., and J. Zhu (1991). The three R's of engineering analysis and error estimation and adaptivity, *Computer methods in Applied Mechanics and Engineering* **82**(1-3), 95-113.

Zulawinski B. (1995). The Swapping Heuristic for Partitioning Problems, MSc thesis, Dept. of Computer Science, Michigan State University, August '95.

Honor Award

to

George Bilchev

For Outstanding Achievement

The Winner of the First ICEO Competition

1996 IEEE International Conference on Evolutionary Computation (ICEC'96)

May 22, 1996
date

Toshio Fukuda
Toshio Fukuda
Conference General Chair

IEEE Neural Network Council (NNC)
Society of Instrument and Control Engineers (SICE)

The 5th Annual Conference on Evolutionary Programming Best Student Paper Award

Presented March 1, 1996 to:

George Bilchev

"Evolutionary Metaphors for the Bin Packing Problem"

John R. McDonnell, President, EP Society

Lawrence J. Fogel, General Chairman, EP96

The Ant Colony Metaphor for Searching Continuous Design Spaces

G BILCHEV[1] and I.C. PARMEE[2]

[1,2] *Plymouth Engineering Design Centre, University of Plymouth*
email: GBilchev@plymouth.ac.uk

Abstract

This paper describes a form of dynamical computational system—the ant colony—and presents an ant colony model for continuous space optimisation problems. The ant colony metaphor is applied to a real world heavily constrained engineering design problem. It is capable of accelerating the search process and finding acceptable solutions which otherwise could not be discovered by a GA. By integrating the Pareto optimality concept within the selection mechanism in GAs and Ant Colony it is possible to treat both hard and soft constraints. Hard constraints participate in a penalty term while soft constraints become part of a multi-criteria formulation of the problem.

Keywords: artificial ant colony, co-operative searches, dynamical computational systems, evolutionary computing, genetic algorithms

1. Introduction

A great majority of natural and artificial systems are of complex nature, and scientists choose more often than not to work on systems simplified to a minimum number of components in order to observe "pure" effects. An alternative approach, often known as the *complex systems dynamics* approach [G.Weisbuch], is to simplify as much as possible the components of the system, so as to take into account their large number. This idea has emerged from a recent trend in research known as the *physics of disordered systems*.

Complex dynamic systems in general show interesting and desirable behaviour as *flexibility* (in vision or speech understanding tasks the brain is able to cope with incorrect, ambiguous or distorted information, or even to deal with unforeseen or new situations without showing abrupt performance breakdown), *robustness* (keep functioning even when some parts are locally damaged), and they operate in a *massively parallel fashion*. Systems of this kind abound in nature. A vivid example is provided by the behaviour of a society of termites [P.J.Courtois].

> While individual termites are only able to perform very simple tasks such as transporting and dropping small quantities of earth in a quasi random fashion, an entire society of these insects is capable of building large nests with a sophisticated structure. Complex and organised behaviour thus emerges out of the massively parallel interactions between the many simple

members of the society. Moreover, this behaviour also shows flexibility and robustness. Flexibility, because the society is able to operate under very different circumstances, depending on the environment and on the structure of the subsoil. Robustness, because we can easily remove a number of termites without touching the society's nest building capabilities.

Complex dynamical systems show *emergent properties*. This means that the behaviour of the system as a whole can no longer be viewed as a simple superposition of the individual behaviours of its elements, but rather as a side effect of their collective behaviour. Contained in this notion is the idea that properties are not *a priori* predictable from the structure of the local interactions and that they are of functional significance.

Complex dynamical systems used for computation are called *dynamical computation systems*. The computation to be performed is contained in the dynamics of the system, which is determined by the nature of the local interactions between the many elements.

Many of the dynamical computation systems that have been developed today find their equivalent in nature. Examples include genetic algorithms [J.H.Holland], spin glass models [W.Kinzel], connectionist architectures [D.E.Rumelhart], reaction-diffusion systems [L.Steels] and simulated annealing [S.Kirkpatrick].

An important notion in dynamical computational systems is that of *interaction*. We concentrate on a particular kind of interaction: that of *co-operation*. Co-operation involves a collection of agents that interact by communicating information, or hints (usually concerning regions to avoid or likely to contain solutions) to each other while solving a problem. The information exchanged may be incorrect at times and should alter the behaviour of the agents receiving it. An example of co-operative problem solving is the use of the *genetic algorithm* to find states of high fitness in some space. In a genetic algorithm members of a population of states exchange pieces of themselves or mutate to create a new population, often containing states of higher fitness. Another example is *neural networks*, where the output of one neuron affects the behaviour of the neuron receiving it.

In this paper we concentrate on a particular type of computational task, that of *search*, which arises for problems with no known algorithmic method for direct solution construction.

Co-operative search methods are based on modifying individual search methods. A useful distinction is whether a method is *complete* or *incomplete*. Complete methods systematically examine states and are guaranteed to either eventually find a solution or terminate when no solution exists. By contrast, incomplete methods explore more opportunistically and may miss some states in the search space, hence they can never guarantee a solution does not exist. For parallel searches, a further

issue is whether to split the search space among the agents. In the simplest case, each agent examines the entire search space. However this can mean a single state is examined by more than one agent during the search. This can be avoided by partitioning the search space into disjoint parts and assigning one to each agent. In this partitioned search, agents only examine states in their assigned part of the space thus avoiding unnecessary duplicate examination of states. Restricting each agent to examine a state at most once, as well as partitioning the search space so that a state is not examined by more than one agent, may improve performance somewhat, but far less than the enhancement achieved by co-operation [S.H.Clearwater].

This paper describes a particular kind of dynamical computational system—that of the ant colony. We review results on modelling ant colonies for order based problems and then propose a model for continuous space optimisation.

2. The Ant Cycle Algorithm for order based problems

Problems like the Travelling Salesman Problem (TSP) and Bin-packing can be represented as a sequence of n items (n cities to be visited or n objects to be packed), where the actual order of the sequence determines a particular solution to the problem. Thus in general the search space consists of all $n!$ permutations. For such order based representations it is natural to apply the ant colony metaphor. In the following paragraphs the TSP [A. Colorni] will be considered, because of its default interpretation of the items as cities, e.g. locations on a 2D map. Extensions to the Bin-packing problem and other order based representations will also be given

For TSP with n cities the objective function is:

$$f(\pi) = \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)})$$

where $d(c_i, c_j)$ is the distance between cities i and j, and $\pi(i)$ for $i=1,n$ defines a permutation. Let m be the number of ants. Then

$$m = \sum_{i=1}^{n} b_i(t)$$

where $b_i(t)$ is the number of ants in city i at time t.

There is also a global structure that represents the nest neighbourhood. In terms of the TSP it represents the distance between each pair of cities and the trail τ_{ij} left by the ants in the course of the algorithm execution. The trail is defined by:

$$\tau_{ij}(t+n) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t, t+n)$$

where ρ is an evaporation constant, t is the beginning of a tour, and $t+n$ is the beginning of the next tour.

$$\Delta\tau_{ij}(t,t+n)=\sum_{k=1}^{m}\Delta\tau_{ij}^{k}(t,t+n)$$

$$\Delta\tau_{ij}^{k}(t,t+n)=\begin{cases}Q/L^{k} & \text{if } (i,j) \text{ is in the tour of ant } k\\ 0 & \text{otherwise}\end{cases}$$

In general Q/L^k is proportional to the success (fitness) of ant k, where Q is a constant and L^k is the tour length of the k-th ant.

Then during the next $(t+n)$ tour the probability to visit city j when being at city i is

$$P_{ij}(t)=\begin{cases}\dfrac{[\tau_{ij}(t)]^{\alpha}\cdot[\eta_{ij}(t)]^{\beta}}{\sum_{k\in allowed}[\tau_{ik}(t)]^{\alpha}\cdot[\eta_{ik}(t)]^{\beta}} & \text{if } j\in allowed\\ 0 & \text{otherwise}\end{cases}$$

where *allowed* is a set of cities not visited for that particular tour. η_{ij} is local heuristics. For the TSP η_{ij} may be the greedy choice, e.g.

$$\min_{j} d(i,j)$$

Now the extension for the Bin-packing problem seems easy. Only the objective function is different, representing the nature of the problem. A good objective function is:

$$f(\pi)=B+\alpha/\mathrm{var}_{empty\_space}$$

where B is the number of boxes in the solution, $\mathrm{var}_{empty\_space}$ is the variance of the empty space of each box, and α is an appropriate coefficient representing a trade-off between the two criteria: $\mathrm{var}_{empty\_space}$ and B. In practice α gives total preference to B and $\mathrm{var}_{empty\_space}$ is used only to differentiate between two solutions with equal B's. If the local heuristics η_{ij} cannot be defined, then η_{ij} can be assumed to be equal to 1 for all ij.

The algorithm runs as follows: First P_{ij}'s are randomly initialised for some small values and m ants are allowed to make a tour. Then the solutions are compared and trail is laid proportionally to the ants' fitness. This alters the P_{ij} values so that on the next tour the probability of repeating (part of) previous good tours increases. This is reminiscent of *schemata propagation* in Genetic Algorithms where building blocks of high fitness pass from one agent to its offspring.

3. The Ant Colony Metaphor for Continuous Spaces

When the search space is continuous the ant cycle algorithm cannot be applied unless some kind of an order based representation is invented. Maintaining analogy with the foraging strategies of ant colonies we suggest a model applicable to continuous spaces. The main difficulty is how to model a continuous nest neighbourhood with a discrete structure. We have achieved this by representing a finite number of directions as vectors starting from a base point (nest). As we potentially have to cover all of the continuous space neighbourhood, these vectors are evolving in time according the ants' fitness (Fig.1).

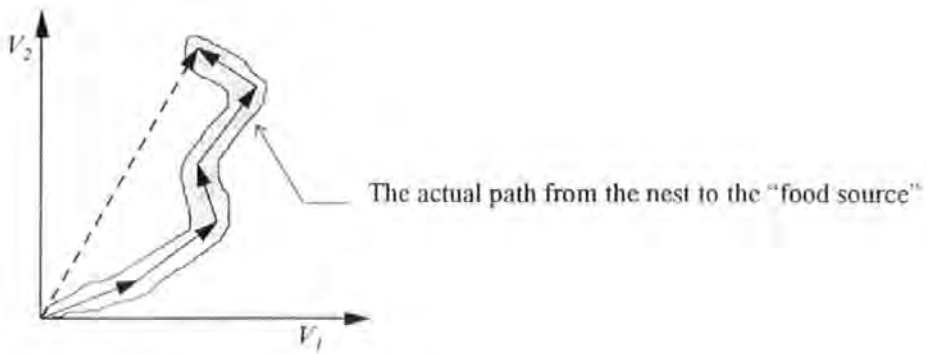


Fig. 1: A vector representing the actual path between the nest and the "food source" after five steps.

```
procedure Ant Colony Algorithm
begin
  t ← 0
  initialize A(t)
  evaluate A(t)
  while (not termination_condition) do
  begin
        t ← t + 1
        add_trail A(t)
        send_ants A(t)
        evaluate  A(t)
        evaporate A(t)
  end
end
```

Fig. 2: The structure of the Ant Colony Algorithm. `A(t)` is a data structure representing the nest and its neighbourhood.

The Ant Colony Algorithm

The structure of the Ant Colony Algorithm is shown in Fig. 2. Before the algorithm begins we have to determine the location of the nest. It should be a point in the search space which seems promising for fine local search exploitation. We suggest finding it by utilising a niching GA or a related strategy. Next we define a search radius R, which determines the extent of the subspace to be considered in each generation (cycle). Then `initialize A(t)` sends ants in various directions at a radius not greater than R (see Fig.3); `evaluate A(t)` is a call of the objective function for all ants; `add_trail A(t)` is proportionally (to the ants' fitness) adding trail quantity to the particular directions the ants have selected, `send_ants A(t)` sends ants by selecting directions using a Roulette wheel selection on the trail quantity and making a random step from the location of the best previous ant that have selected the same direction (see Fig.4), `evaporate A(t)` is decrementing the trail. The random step is implemented as

$$\Delta(t,R) = R \cdot (1 - r^{(1-t/T)^b})$$

where R is the search radius, r is a random number from [0..1], T is the maximal generation number, and b is a system parameter determining the degree of non-uniformity. $\Delta(t,R)$ returns a value in the range $[0..R]$ such that the probability of $\Delta(t,R)$ being close to 0 increases as t increases. R is determined by the extent of the search subspace we want to cover during the run.

If certain directions do not result in improvement, they do not participate in the trail adding process and the reverse (evaporation) process diverts attention away from them. This can be thought of as an analogy of a food source exhausting.

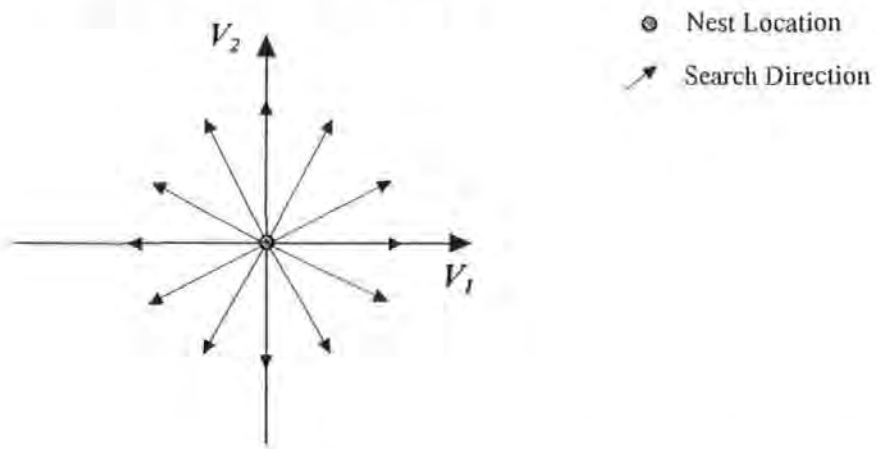
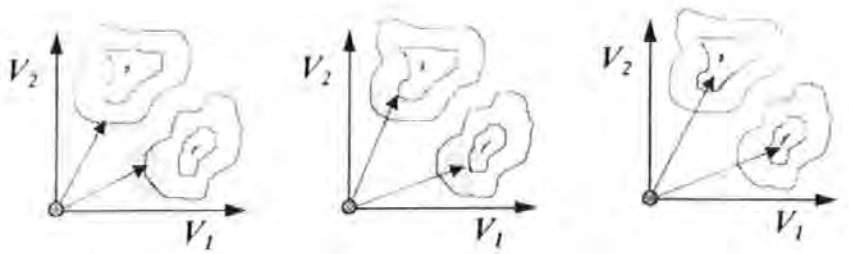
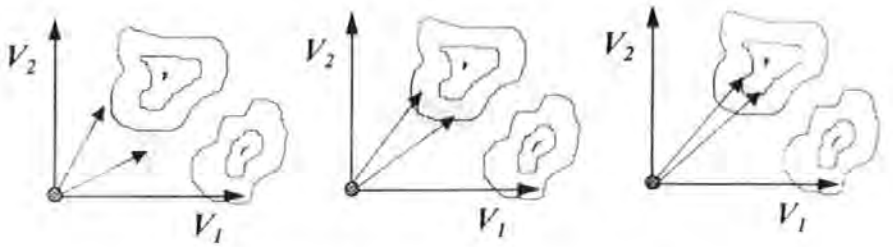


Fig. 3: Two dimensional Nest neighbourhood model with twelve search directions. Each direction evolves in time according to the fitness of the ants that have selected it. Typical direction evolution is shown in Fig. 4.



a) different basins of attraction for the two directions



b) the same basin of attraction for the two directions

Fig 4: Evolution of directions. The shaded region shows the search radius R at each step.

To make the model more accurate a random walk and trail diffusion can be added. The basic trail diffusion idea is shown on fig. 5. It is important to notice the analogy between trail diffusion in the Ant Colony and arithmetic crossover in GAs. In arithmetic crossover the new chromosome is constructed from its parents by a linear combination of the individual parameters: $\alpha_i x_i^1 + (1-\alpha_i) x_i^2$. The distribution of α_i can be assumed normal with mean value equal to $0.5(x_i^1 + x_i^2)$ and the probability of generating new offspring with particular parameter values can be represented as contour plot in the form of concentrated hyperspheres, i.e., closer to the centre, greater the probability (fig. 5a). In the Ant Colony the intersection of the diffused trail from two different paths form a virtual path with a location probability determined by the superposition of the diffused trail strength (fig. 5b).

An obvious limitation of the current algorithm is the lack of a model for the exhausting of the food source. Thus it is possible for search agents to repeat already tracked and assumed exhausted paths.

4. Experimental Results

The Ant Colony model is used in a real world heavily constrained engineering design problem. The domain involves preliminary air-frame design and the definition of a flight trajectory for an air-launched winged rocket that will achieve

orbit before returning to atmosphere for a conventional landing. The problem is extremely sensitive to five non-explicit (embodied in a simulation program) non-linear constraints relating to air-speed and climb angle. Constraints also affect the physical parameters describing the air-frame and engine configuration. The problem is to minimise the empty weight of the vehicle through a space of seven continuous and one discrete design variables, subject to these five non-linear constraints. Initial discussion revealed a degree of doubt as to whether a feasible solution to the problem actually exists and preliminary experimentation using a GA with the constraint violation as a fitness function could only achieve solution exhibiting minimum constraint violation.

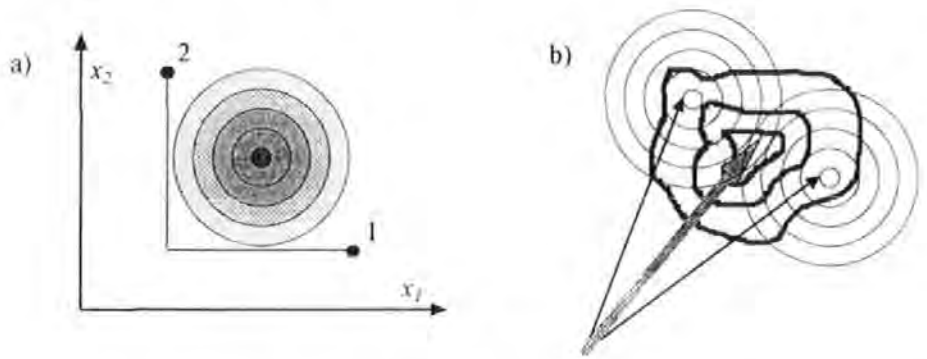


Fig. 5: (a) Contour plot of the probability for generating new offspring from two parents (1 and 2) by arithmetic crossover in GAs. (b) Superposition of trail diffusion (the concentrated circles) forms a kind of virtual path (the grey vector). The graph is superimposed over the contour plot of the fitness function to show the expected effect of trail diffusion.

A GA with floating point representation is used because it offers (1) a significant reduction in the length of the chromosome, (2) an ability to express and operate on parameters in their natural base 10 encoding, and (3) ease in interfacing to other algorithms (e.g. standard numerical optimisation techniques, regression analysis, etc.). A rank based selection scheme with linear normalisation is utilised.

It is evident that a secondary search process is required. As the fitness landscape is still very complex and detailed even when considering small neighbourhoods (0.01% of the parameters range) we cannot use a hill climber search as it will get stuck in local optima. The Ant Colony was initially designed for fine local search applied after the GA has found promising clusters for future exploration, although some preliminary results show it can also search well in larger spaces.

A particular run of the GA is shown in Fig. 6. Fig. 7 shows the result obtained when applying the Ant Colony from the point found by the GA in Fig. 6.

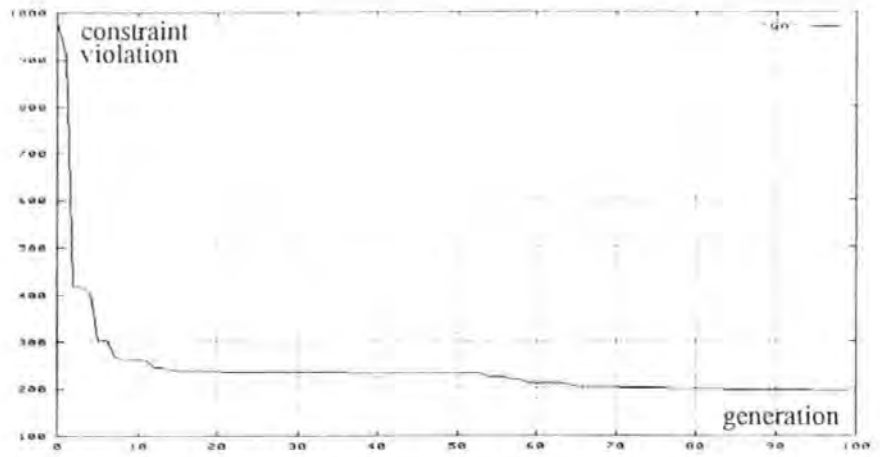


Fig. 6: A particular run of a GA for 100 generations. The graph shows the distance from the constraint values defined as the sum of the squares of the difference between each current value of the particular constrained variable and its constrained value.

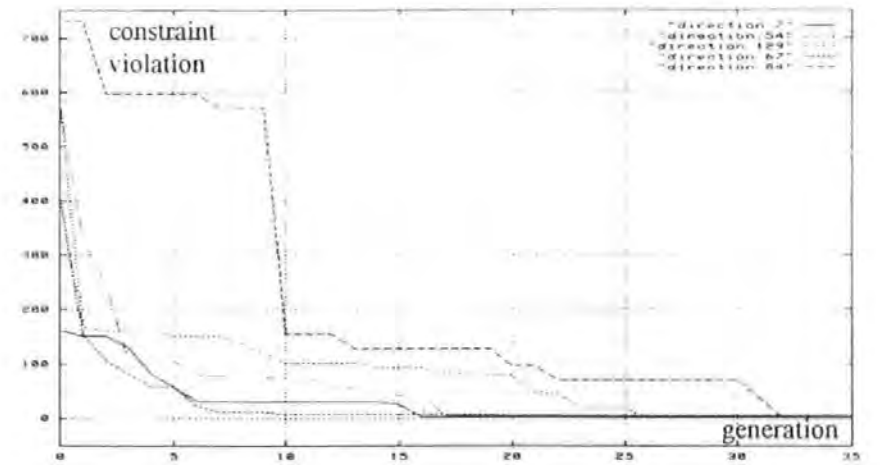


Fig. 7: The Ant Colony Algorithm applied on the point found by the GA from Fig. 6. The evolution of the distance from the constrained values of five directions is shown.

Current work at the **Plymouth Engineering Design Centre** [G. Bilchev, and I.C. Parmee] involves the integration of the Pareto optimality concept into the selection mechanism of GAs and Ant Colony. The main motivation is that standard non-linear programming methods cannot treat both soft and hard constraints. Previous work within the Centre has addressed the representation of soft constraints as multi-

objectives by utilising a modified VEGA approach [I.C. Parmee, and G. Purchase]. A Pareto approach will also allow soft constraints to be incorporated as multi-criteria while hard constraints can be treated in the usual way in penalty terms.

5. Sensitivity Analysis

A sensitivity measure is defined in terms of maximum and average risk to achieve degraded real design when deviating from the numerically represented design solution. That risk is unavoidable because of the physical incapability to achieve the exact values of the design variables. It may happen that even when using numerically stable algorithms the found optimal design solution lies within a very sensitive region and a small perturbation in the design variables can lead to an enormous change in the overall design solution [I.C. Parmee and M.J. Denham]. This is a property of the problem itself and does not depend on the actual optimisation algorithm. So when making decisions the engineering designer, among others, should take into consideration the sensitivity of a given solution.

A generic method for calculating the sensitivity is presented, which is applicable to problems with non-explicit objective and constraint functions. The proposed sensitivity measure is also capable of representing design variables interaction. The price paid for that is increase in the number of the objective function calls. A method for incorporating sensitivity analysis with the search process and thus essentially reducing the number of the objective function calls is proposed.

5.1 Definition

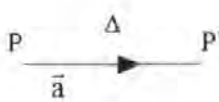
Let's denote our objective function by $F: \Re^n \rightarrow \Re$ or $F(x_1, x_2, \ldots, x_n)$. Then F can be considered a scalar field and the gradient is defined by:

$$\text{grad } F = \frac{\partial F}{\partial x_1} \cdot \vec{i}_1 + \frac{\partial F}{\partial x_2} \cdot \vec{i}_2 + \cdots + \frac{\partial F}{\partial x_n} \cdot \vec{i}_n$$

where $\vec{i}_1, \ldots, \vec{i}_n$ are the unit vectors of an orthogonal co-ordinate system. The gradient at each point has a length and direction that is independent of the particular choice of Cartesian co-ordinates. If at a point **P** the gradient of F is not the zero vector, it has the direction of the maximum increase of F at **P**.

The directional derivative is defined as:

$$D_a F = \frac{dF}{d\Delta} = \lim_{\Delta \to 0} \frac{F(P') - F(P)}{\Delta}$$



$P = (x_1, \ldots, x_n)$

$P' = (x_1 + \Delta x_1, \ldots, x_n + \Delta x_n)$

and is effectively calculated by:

$$D_{\vec{a}}F = \frac{1}{|\vec{a}|}\ \vec{a} \bullet \text{grad}\ F = |\text{grad}\ F| \cdot \cos\alpha$$

where α is the angle between $\vec{a}$ and grad F at a particular point.

Using finite differences the directional derivative can be approximated by:

$$D_{\vec{a}}F = \frac{dF}{d\Delta} \approx \frac{F(P') - F(P)}{\Delta}$$

for some small positive real value Δ.

If the gradient has the direction of the maximum increase of F at **P** then:

$$\vec{G}(x_1^P, \ldots, x_1^P) = \text{grad}\ F = \lim_{\Delta \to 0} \left(\frac{\vec{P}' - \vec{P}}{|\vec{P}' - \vec{P}|} \cdot \max_{\forall P'} \frac{F(P') - F(P)}{\Delta}\right)$$

Now a *neighbourhood sensitivity* can be defined as:

$$\vec{\hat{S}}_{max}(\delta) = \vec{\hat{G}}(x_1^P, \ldots, x_n^P) = \left(\frac{\vec{P}' - \vec{P}}{|\vec{P}' - \vec{P}|} \cdot \max_{\forall P'} |F(P') - F(P)|\right)$$

$\vec{\hat{S}}$ is an estimation of the maximum degradation that can be achieved in the real design. In a similar manner an estimation of the average degradation is defined:

$$\vec{\hat{S}}_{mean}(\delta) = \vec{\hat{G}}(x_1^P, \ldots, x_n^P) = \left(\frac{\vec{P}' - \vec{P}}{|\vec{P}' - \vec{P}|} \cdot \text{mean}_{\forall P'} |F(P') - F(P)|\right)$$

The neighbourhood sensitivity is a vector such that its magnitude represents the difference of the fitness function calculated at two points (**P'** and **P**) while its direction gives information about the actual design variables influence on that fitness function change.

5.2 Calculating Sensitivity

The definitions of $\vec{\hat{S}}_{max}$ and $\vec{\hat{S}}_{mean}$ imply the algorithms for their calculation. Finding $\vec{\hat{S}}_{max}$ is a search process on the surface of a hypersphere defined by

$(x_1^{P'} - x_1^P)^2 + \cdots + (x_n^{P'} - x_n^P)^2 = \delta^2$ while finding $\bar{\bar{S}}_{mean}$ is sampling that hypersphere and calculating the average |F(P')−F(P)|.

The Ant Colony algorithm is adopted to find $\bar{\bar{S}}_{max}$ because it is a robust multi-modal search technique that will give information about various risky directions of degradation. Moreover the intention is to later avoid a separate search for calculating the sensitivity by integrating it with the second phase of the overall design solution search process. This can be implemented by keeping track of all model evaluations during that second phase Ant Colony search and then after a solution is accepted the sensitivity can be calculated by referencing the previous model call results. It may happen that in certain directions the hypersphere is not well sampled which will require some extra model calls. It is recommended that the sensitivity is now calculated through a hypersphere layer defined by:

$$(\delta - \varepsilon)^2 \le (x_1^{P'} - x_1^P)^2 + \cdots + (x_n^{P'} - x_n^P)^2 \le (\delta + \varepsilon)^2$$

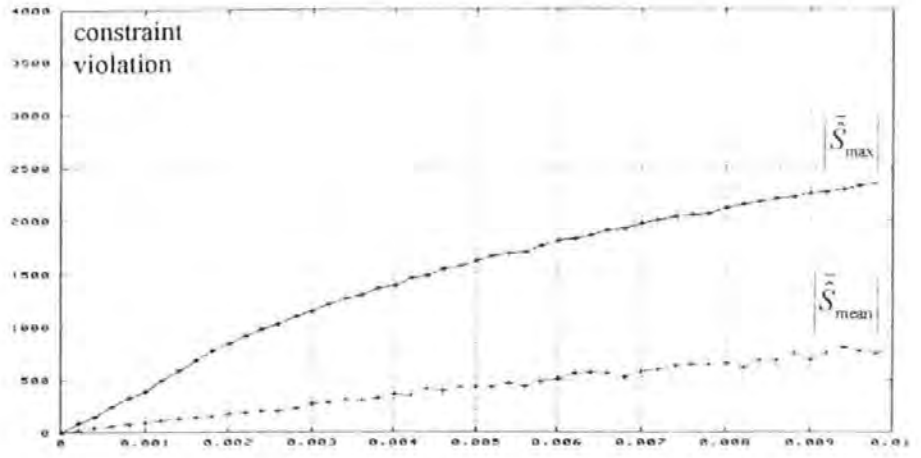
where ε defines the thickness of the layer. The graph now will be more like a histogram.

5.3 Experimental Results

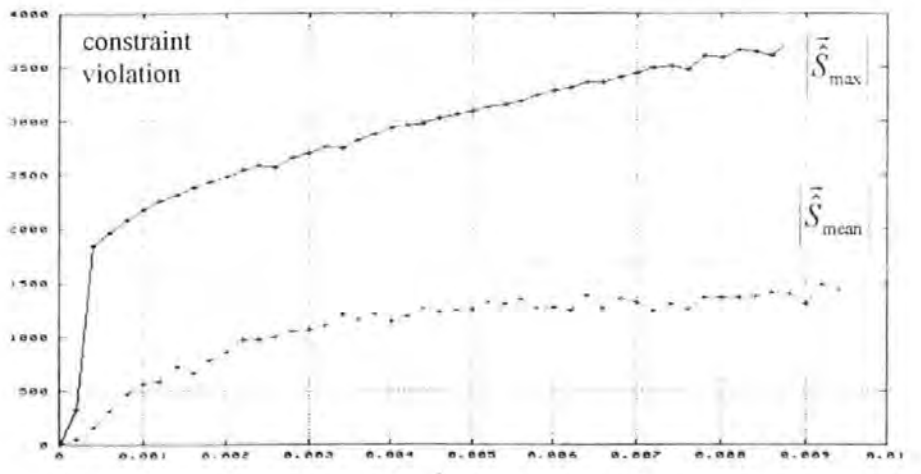
The sensitivity is calculated at two points found by the search procedure described in section 4. Considering that both points seem to be promising design solutions according to the constraint and/or multi-criteria satisfaction the sensitivity can be used as an additional decision criterion. Fig. 8 clearly shows that the solution at points 1 is less sensitive than the solution at point 2. The engineering designer may also want to take into consideration the degree to which each design variable contributes to the sensitivity. Table 1 shows that information for point 1.

δ%	Δα	Δγ	...	Δh_{MECO}
0.0001	-0.51527	0.50107	...	-0.15359
0.0002	-0.48958	0.47331	...	-0.01903
0.0003	0.56719	-0.41718	...	+0.02065
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
0.0099	-0.38798	0.33810	...	-0.06375
0.0100	-0.28638	0.25271	...	-0.07806

Table 1: Direction information of the maximum sensitivity for point 1 (fig. 8). The numbers in the table define a direction of a unit radius vector in a hyperspace.



point 1



point 2

Fig. 8: Sensitivity calculated at two points of the search space

5. Discussion

The Ant Colony dynamics provide a control for the trade-off between exploration and exploitation during a search process. In the present paper we have used it with a simple hill climbing algorithm in order to show the power of co-operation, but it can be integrated in a similar way with many search techniques. We have found several advantages in the Ant Colony Metaphor used for finding good feasible solutions:

- The scope of the feasible region can be outlined as the population represents the best individuals in various directions. (Sometimes it is possible for some of the initial direction information to be lost due to migration of agents from one direction to another (see Fig. 4b), but this can be controlled by parameters of the Ant Colony dynamics.)
- Many search directions are considered in parallel
- Easy to integrate with many search techniques (hill climbers, GA's, etc.)
- Better than local search if the search space contains numerous basins of attraction. (We also believe it is better than local search for long path problems [J.Horn, D.Goldberg].)

We have become more convinced that rather than spending all the effort in developing a monolithic program or perfect heuristic, it may be better to have a set of relatively simple co-operating processes working concurrently on the problem while communicating their partial results.

Acknowledgements
This research is supported by the Plymouth Engineering Design Centre at the University of Plymouth. British Aerospace has provided experimental design software and guidance. We thank these organisations for their continuing support.

References

D.E.Rumelhart and J.L.McClelland, Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1, Bradford Books, Cambridge, MA

Deniz Yuret, and Michael de la Maza, Dynamic Hill Climbing: Overcoming the limitations of optimization techniques. Technical report, Numinous Noetics Group, Artificial Intelligence Laboratory, MIT, MA 02139

George Bilchev, Evolutionary Algorithms for the Bin-packing Problem, Chapter 4, Comparison between Ant Colony Search and Genetic Algorithms. *MSc thesis*, 1994, New Bulgarian University, Sofia 1125, Bulgaria (in Bulgarian)

George Bilchev, and I.C. Parmee, Searching Heavily Constrained Design Spaces, *Procs. of the 22nd International Conference on CAD-95*, 8-13 May, 1995, Yalta, Ukraine.

George Bilchev, and I.C. Parmee, Natural Self-organizing Systems, Internal Report PEDC-03-95, Engineering Design Centre, University of Plymouth, UK

Gerard Weisbuch, Complex Systems Dynamics, Lecture Notes Volume II, Santa Fe Institute, Studies in the Sciences of Complexity, Addison-Wesley, 1991

I.C.Parmee, and G.Purchase, The Development of a Directed Genetic Search Technique for Heavily Constrained Design Spaces, in *Proc. of Adaptive Computing in Engineering Design and Control*, Plymouth, 1994

I.C. Parmee and M.J. Denham, Emergent Computing Methods in Engineering Design, NATO Advanced Research Workshop, Nafplio, Greece, August 1994.

J.H.Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications in Biology, Control, and Artificial Intelligence, University of Michigan Press, Ann Arbor, 1975

J.Torreele, Optimization by Simulated Annealing. Introduction and Case Study, AI-MEMO 88-18, AI-lab VUB, Brussles

Jeffray Horn, N. Nafpliotis, and D.E.Goldberg, A Niched Pareto Genetic Algorithm for multiobjective optimization, in *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, Volume 1*, 1994, Piscataway, NJ

Jeffrey Horn, David E. Goldberg, and Kalyanmoy Deb, Long Path Problems, *Proceedings of Parallel Problem Solving from Nature*, Lecture Notes in Computer Science, Springer-Verlag, 1995

L.Nadel and D.Stein, eds., Better than the Best: The Power of Cooperation, *Complex Systems*, 163-184, Addison-Wesley 1993

L.Steels, Artificial Intelligence and Complex Systems, AI-MEMO 88-2, AI-lab VUB, Brussels

Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo, An investigation of some properties of an "Ant algorithm", in *Procs. of the PPSN '92*, Elsevier Publishing, pp 509-520

P.J.Courtois, On line and Space Decomposition of Complex Structures, *Comm. of the ACM*, Vol.28, no.6

S.Kirkpatrick, Gelatt C.D., M.P.Vechi, Optimisation by Simulated Annealing, *Science*, Vol.220, No.4598, May, 1983

Scott H. Clearwater, Bernardo A. Huberman, and Tad Hogg, Cooperative Problem Solving, in B. Huberman, editor, *Computation: The Micro and the Macro View*, pages 33-70, World Scientific, Singapore, 1992

W.Kinzel, Learning and Pattern Recognition in Spin Glass Models, 1985

Proceedings of the. 22nd International. Conference CAD '95, Ukraine, Yalta, May 8-13, 1995

ADAPTIVE SEARCH STRATEGIES FOR HEAVILY CONSTRAINED DESIGN SPACES

G. Bilchev and I. C. Parmee
Plymouth Engineering Design Centre, University of Plymouth, PL4 8AA, UK
email: GBilchev@plymouth.ac.uk

ABSTRACT

This paper describes a multi-stage evolutionary methodology for searching heavily constrained design spaces. It utilises the Genetic Algorithm (GA) as a pre-processor for allocating promising solution clusters followed by an Ant Colony (AC) model to provide fine-grained localised search. The methodology is currently refined to deal with both soft and hard constraints by integrating the concept of Pareto optimality within the selection mechanism of the evolutionary system. The first stage of the search is controlled by an automated Decision Maker (DM) that incorporates the prior constraint satisfaction preferences. In the final stage of the search process the Engineering Designer (ED) is actively included, thus exploiting his implicit expertise. The methodology is successfully applied on a real world heavily constrained engineering design problem provided by British Aerospace plc., UK.

1. INTRODUCTION

Standard non-linear programming methods do not possess the characteristics of Adaptive Search techniques in that they fail to make use of or learn from information generated at previous stages in the search [1]. This is very different from the iterative, manual design performed by an engineer who is constantly gathering and updating information throughout the process. Virtually all successful algorithms make use of gradient information to guide the search and as such they tend to be trapped by local optima.

Where the search is hampered by high dimensionality or non-linearity more "global" search techniques, such as the GA, have proven useful. A primary characteristic of GA search is the algorithm's ability to sample widely varying areas of highly dimensional spaces [2]. In many cases this characteristic has the potential to offer the engineering designer a high level of decision support by providing a number of diverse, high performance solutions to any design problem. This allows the exploration of regions that may offer innovative yet practical design solutions. Such solutions may otherwise be inaccessible within an engineer's design time and budget constraints [3]. However, this approach will not be appropriate in all cases. In many situations decisions concerning certain design aspects will already be in place. For instance, the use of a specific material may be desirable to satisfy financial and manufacturing criteria. This upstream choice may limit the search process to specific regions of the overall design space and the bounds of these regions may be largely unknown. Therefore, a technique that will direct the search to specific points within a design space is required and the integration of such a method with the GA search engine shows significant potential. This suggests that some modifications should be made to the GA framework in order to be accepted as a primary and versatile tool in engineering design.

2. CONSTRAINTS AND GENETIC ALGORITHMS

The Genetic approaches to constrained optimisation mainly fall into three categories: (a) *modification of representation and genetic operators*, (b) *penalty functions*, and (c) *direct Pareto techniques*.

The modification of representation and genetic operators concentrates on using special representation decoders which guarantee (or at least increase the probability of) the generation of a feasible solution and on the application of special repair algorithms to "correct" any infeasible solution so generated. However, decoders are frequently computationally intensive to run, not all constraints can be easily implemented this way, and the chances of building a general genetic algorithm to handle different types of constraints based on this principle seem to be slim. The GENOCOP approach [4] is based on an elimination of the equalities present in the set of

constraints and the careful design of special "genetic" operators which guarantee to keep all "chromosomes" within the constrained solution space. The approach is applicable to linear constraints only and cannot be generalized for non-convex search spaces. These approaches are not appropriate for heavily constrained problems where the work involved in finding an initial population of feasible solutions is considerable. It seems clear that some sort of relaxation of the constraints may be necessary in order to allow the genetic search to proceed at a reasonable pace. One solution of the problem is by using penalty functions.

A constrained problem is transformed to an unconstrained problem by associating a penalty with all constraint violations and the penalties are included in the function evaluation. If the penalties are used within a genetic approach it is not essential for the penalty term to have any particular form, such as being unimodal or smooth, beyond having a fitness function that is easily evaluated [5]. The difficulty of using penalties within GAs stems from the fact that in the context of highly constrained optimisation an infeasible solution with strong genotypic similarity to the optimal constrained solution is more useful in an intermediate population than is a feasible solution with weaker genotypic affinity to the optimum. For instance, an overzealous penalty may reward schemata which quickly, but wastefully satisfy constraints. An over-tolerant penalty function will be unable to provide sufficient pressure to satisfy constraints and non-feasible solutions will be highly fit. To some extent this trade-off problem can be overcome by using dynamic penalty functions, but the difficulties in the application of the dynamic penalty function method are that the exact feasibility/infeasibility trade-off schedule cannot be effectively computed and to the best of the authors knowledge existing solutions are highly problem dependent and can hardly be generalised [6]. An attempt to overcome these difficulties can be found in the Behavioral Memory paradigm [7] where the general problem of genetic constrained optimisation is addressed by a multi-step process: (1) evolve an initial random population with some standard GA, the fitness function being related to the constrained satisfaction, and (2) take the final population resulting from this evolution and use it as an initial population for a GA with the objective cost function as fitness function This is overridden by assigning zero fitness whenever the constraints are not satisfied. Some obvious drawbacks of the Behavioral Memory method are the assumption that the feasible region is large and that the constraints are linearly ordered. For a detailed overview of existing constraint handling techniques in Evolutionary Computation the reader is refered to [8].

For severely constrained problems, where it is extremely difficult to find a feasible solution, the GA approach with penalty functions is not applicable. There are two alternative approaches in solving this problem: (1) developing a hybrid search framework capable of finding feasible regions and (2) treating the constraints as multi-criteria, i.e. utilising direct Pareto techniques.

3. A HYBRID SEARCH FRAMEWORK

In this section we develop a hybrid search framework and test it on a real world severely constrained design problem involving preliminary air-frame design and definition of a flight trajectory for an air-launched winged rocket that will achieve orbit before returning to atmosphere for a conventional landing. The problem is extremely sensitive to five non-explicit non-linear constraints relating to air-speed, climb angle, and physical parameters describing the air-frame and engine configuration. The task is to minimise the empty weight of the vehicle through a space of seven continuous variables and one discrete variable subject to the five constraints. Initial discussion revealed a degree of doubt as to whether a feasible solution to the problem actually exists. Thus the constraints were relaxed and an acceptability notion was defined. Preliminary experimentation using a GA with the constraint violation as a fitness function could only achieve a solution exhibiting minimum constraint violation (fig. 1).

It is evident that further search is required. We define a hybrid search framework (fig. 2) that consists of a floating point GA running for twenty generations followed by an Ant Colony (AC) model. Parmee [9] has shown that a GA with modified selection mechanism and variable mutation is capable of allocating various good design clusters. The cluster information can then be passed to the Engineering Designer (ED) who according to his expertise selects points for

further refinement by the AC. The AC is selected because it is a robust multi-modal search technique relying on multi-agent co-operation in order to distribute search in the most promising areas [10]. Apart from refined solutions the AC also returns sensitivity information that eventually affects the ED final decision.

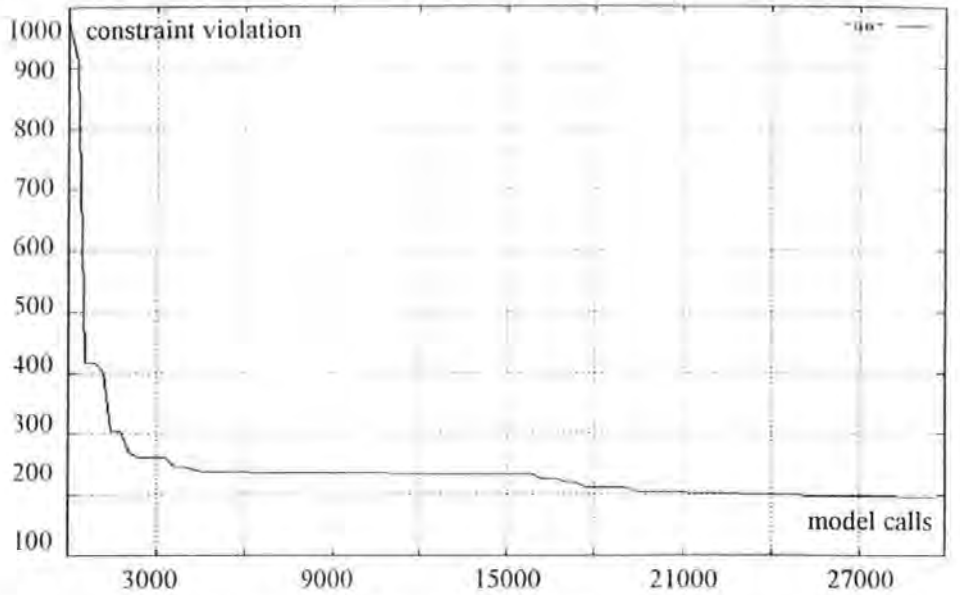


Fig. 1: Result from running a floating point GA with the constraint violation calculated as the Euclidean distance from the acceptable region.

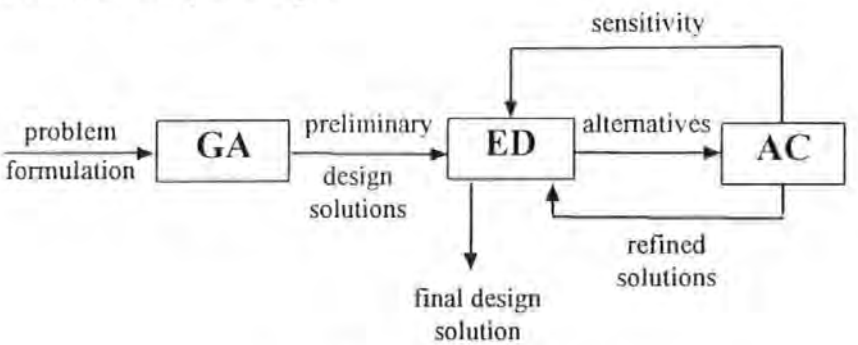


Fig. 2: A hybrid search framework.

The hybrid search framework is now capable of finding numerous acceptable solutions (fig. 3), but still no fully satisfactory solution could be found. A discussion with experts from British Aerospace revealed that the difficulty is in the formulation of the problem which makes it impossible to handle both soft and hard constraints. The only promising strategy for dealing with soft constraints seems to be the utilisation of the Direct Pareto techniques proposed in the following section.

4. DIRECT PARETO TECHNIQUES

4.1 Introduction

Standard techniques for Pareto optimisation of a non-linear vector optimisation problem turn the original problem into a sequence of scalar optimisation problems, which can be solved numerically by applying adapted methods of non-linear programming. Thus, the increase of the computational effort involved in the numerical determination of Pareto optima may prove to be quite considerable, even with problems of moderate size. The Evolutionary systems, however, are readily modified to deal with multiple objectives by incorporating the concept of Pareto optimality into the selection operator and applying a niching pressure to spread the population along the Pareto optimal front. This is usually implemented by adding a Pareto domination tournament [11].

A similar approach for the handling of constraints is proposed in which we consider the constraints to represent individual criteria of a multi-objective formulation of the problem. We shall term this the Direct Pareto approach which has been utilised in previous research within the PEDC as described in section 4.2. In case of equality constraints the distance from the desired constraint values is to be minimised. Inequality constraints are handled by simply optimising the constrained parameter values. When a constraint is not active it turns off the related criteria from the problem formulation. Once feasibility is reached a standard method for constraint handling could be utilised for further search. If the ratio of the feasible region to the overall search space is large a penalty approach [5] would be helpful. However, in the case of a small ratio or numerous isolated feasible regions, analysis of the alternative preliminary solutions may be required. One possible way is to cluster the population and build an Optimal Hypercube (as described in 4.2) around each cluster. A GA with penalties could then be used for the final search within selected hypercubes (fig. 4). Another way is to utilise the Ant Colony algorithm [10] to search within selected clusters.

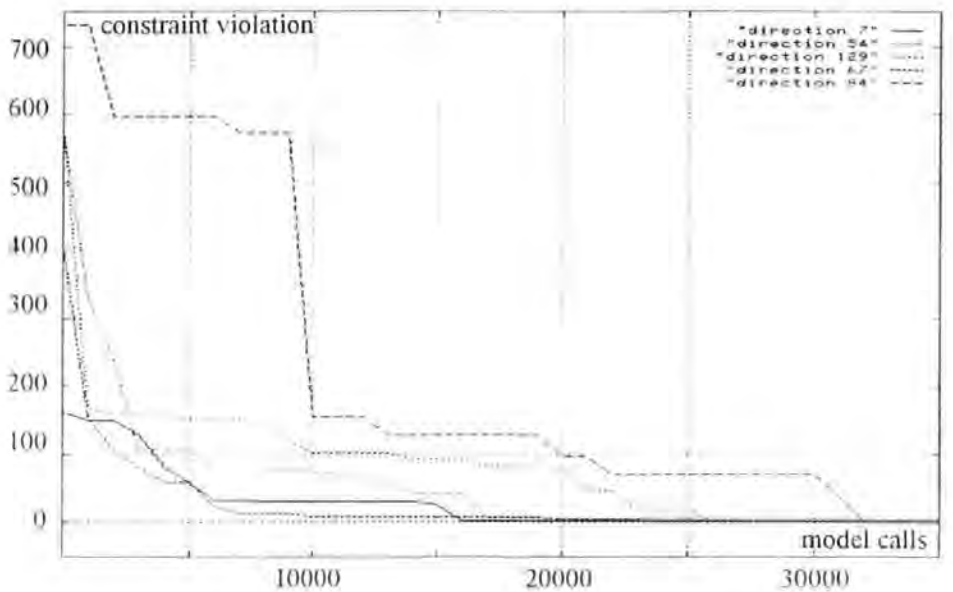


Fig. 3: Result from running the Ant Colony starting at a point found by a GA

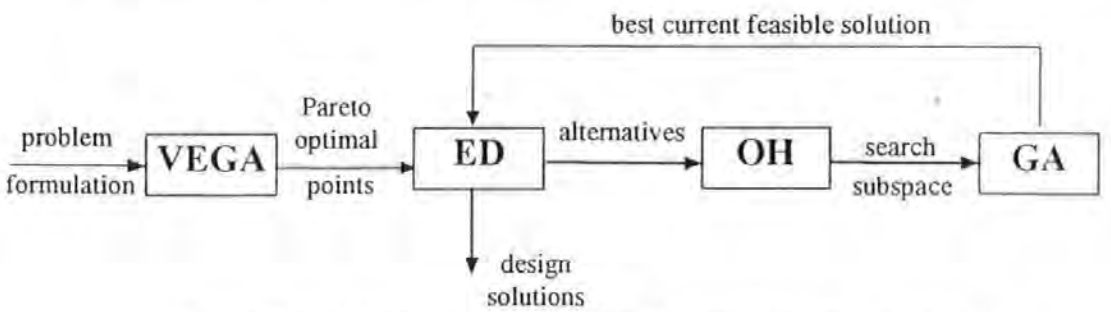


Fig. 4: Search framework utilising the Optimal Hypercube technique

4.2 The Optimal Hypercube (OH) Search Framework

Utopian values within the feasible region are selected and the distance between the constraint and the utopian values [12] provide the fitness function for a modified VEGA search [13]. Then a hypercube that best describes the extent of the feasible region around each VEGA solution is established from a secondary GA search. The fitness of each GA generated hypercube relies upon the degree of constraint violation at particular points upon its surface. A search is then initiated within selected hypercubes to locate an optimal design solution (fig. 4). The whole process can be viewed as problem reduction where the engineering designer is actively involved by making a decision which of any alternative hypercubes should be further

explored by the final search. The Optimal Hypercube search framework is analogous to the Behavioral Memory technique with the advantage of handling both soft and hard non-linearly ordered constraints. The soft constraints are incorporated as directed search criteria, while the hard constraints participate as penalty terms in the fitness function. A limitation of the Optimal Hypercube is the assumption that there exist some sufficiently "large" feasible regions, requiring further search.

5. CONCLUSIONS AND CURRENT RESEARCH

All of the techniques mentioned so far including the Direct Pareto approach, fail to utilise the prior constraint satisfaction preferences. In real world applications, however, the constraint values often have a different degree of acceptability (fig. 5). Therefore, it is more appropriate to guide the search using optimal constraint satisfaction fronts, rather than Pareto optimal fronts. In an attempt to utilise prior constraint satisfaction preferences we conclude by proposing the following search framework (fig. 6).

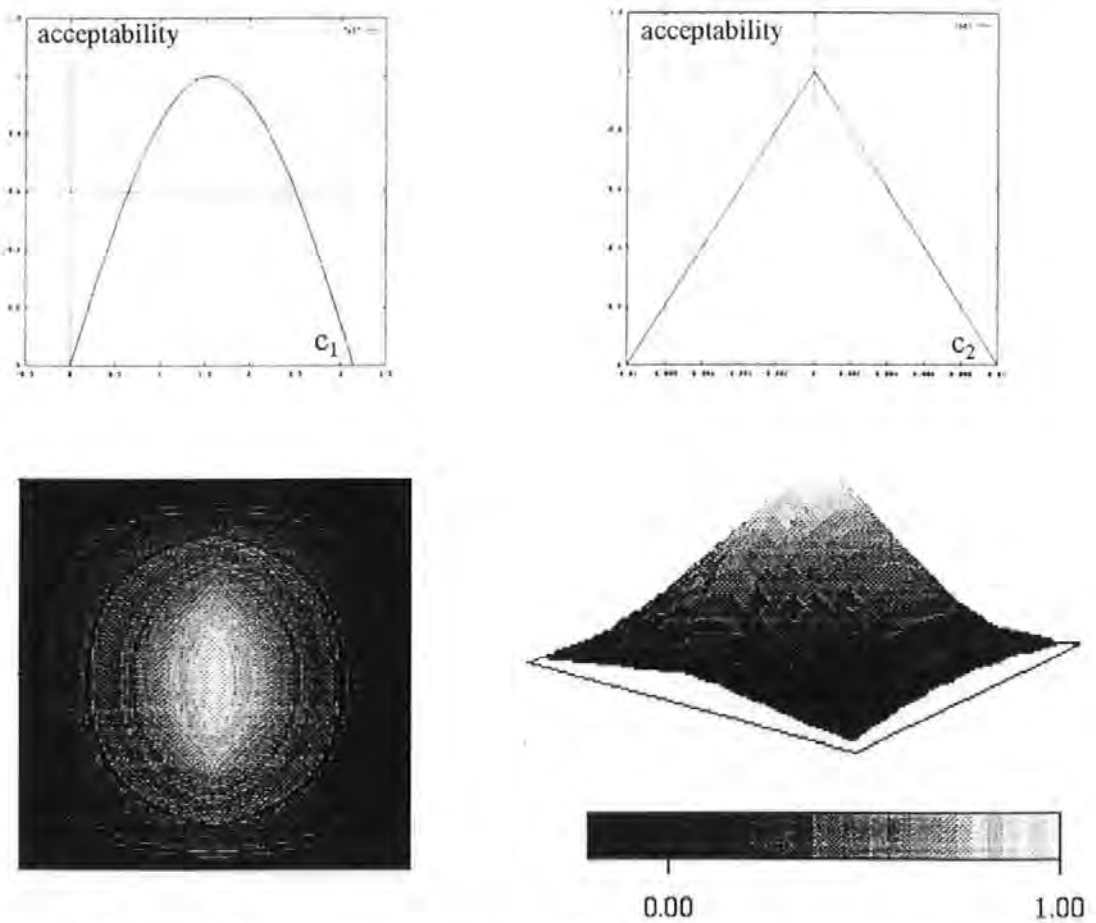


Fig 5: The first pair of graphs shows the degree of acceptability for two constraints. The second pair of graphs shows the constraint satisfaction fronts and the acceptability landscape for the two constraints.

In the first phase of the search process the Decision Maker (DM) incorporates the *a priori* constraint knowledge in terms of an *objective* optimal constraint satisfaction front. The GA and DM are tightly coupled as the DM participates in the selection operator of the GA. The second phase of the search process represents a co-operation between the ED and the AC. The ED uses his expertise to select preliminary solutions for further refinement by the AC and the AC returns sensitivity and performance trade-off information [10] that eventually influences the ED's subsequent choices.

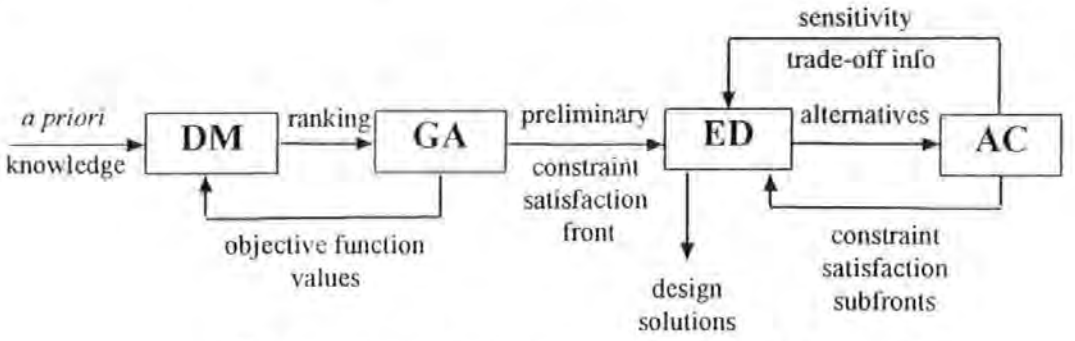


Fig. 6. Proposed structure of a design search process.

ACKNOWLEDGEMENTS

This research is supported by **P**lymouth **E**ngineering **D**esign **C**entre of the University of Plymouth. British Aerospace has provided experimental design software and guidance. We thank these organisations for their continuing support.

REFERENCES

[1] H. Adeli, and K. Balasubramanyam, A Synergic Man-machine Approach to Shape Optimization of Structures, *Computers and Structures*, 30(3), pp. 553-561

[2] D. Goldberg, Genetic Algorithms on Search, Optimization, and Machine Learning, Addison-Wesley Publishing Co., Reading, MA

[3] I. C. Parmee, and M. Denham, The Integration of Adaptive Search with Current Engineering Design Practice, *Proc. of the First International Conference on Adaptive Computing in Engineering Design and Control*, University of Plymouth, 1994

[4] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag, 1992

[5] D. Powell, and M. Skolnick, Using Genetic Algorithms in Engineering Design Optimization with Non-linear Constraints, *Proc. of the 5th International Conference on Genetic Algorithms*, University of Illinois at Urbana-Champaign, 1993, pp. 424-431

[6] J. Richardson, M. Palmer, Some Guidelines for Genetic Algorithms with Penalty Functions, *Proc. of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Los Altos, CA, 1989, pp. 191-197

[7] M. Schoennauer, and S. Xanthakis, Constrained GA optimization, *Proc. of the 5th International Conference on Genetic Algorithms*, University of Illinois at Urbana-Champaign, 1993, pp. 573-580

[8] Z. Michalewicz, A Survey of Constraint Handling Techniques for Evolutionary Computation Methods, The 4th Annual Conference on Evolutionary Programming '95, San Diego, USA

[9] I. C. Parmee, Reinforcing the Natural Clustering Tendencies of the Genetic Algorithm, Internal Report PEDC-04-95; Plymouth Engineering Design Centre, April '95, University of Plymouth, UK

[10] G. Bilchev, I.C. Parmee, The Ant Colony Metaphor for Searching Continuous Design Spaces, Procs. of AISB Workshop on Evolutionary Computing, Univ. of Sheffiled, UK, April'95

[11] J. Horn, N. Nafpliotis, and D. Goldberg, A Niched Pareto GA for Multiobjective Optimization, *1st IEEE Conference on Evolutionary Computation*, 1994

[12] I.C. Parmee, and G. Purchase, The Development of a Directed Search Technique for Heavily Constrained Design Spaces, *Proc. of the First International Conference on Adaptive Computing in Engineering Design and Control*, University of Plymouth, 1994

[13] J. Schaffer, Multiple Objective Optimization with Vector Evaluated Genetic Algorithms, *Proc. of the 1st International Conference on Genetic Algorithms*, 1985

Evolutionary Metaphors for the Bin Packing Problem

George Bilchev

Plymouth Engineering Design Centre
School of Computing
University of Plymouth, PL4 8AA, UK
GBilchev@plymouth.ac.uk

Abstract

An ordering problem, loading objects into boxes, is used as a vehicle for comparing a many-agent search model (MA), a genetic algorithm (GA), and an ant colony search model (AC). Each of the search models is defined and applied to the bin packing problem (BPP). The AC metaphor is generalized to model hybrid distributed cooperative searches. Empirical comparisons reveal the potential power of the hybrid approach when tackling a well-defined search problem.

1. Introduction

1.1 Problem Definition

The bin packing problem (BPP) is defined as follows: given a finite set O of numbers (the object sizes) and two constants C (the bin's capacity) and N (the number of bins), is it possible to pack all the objects into N bins, i.e., does there exist a partition of O into N or fewer subsets, such that the sum of the elements in any of the subsets doesn't exceed C?

This NP-complete decision problem gives rise to the associated NP-hard optimization problem [1]: what is the *best* packing, i.e. what is the *minimum* number of subsets in the above mentioned partition?

Being NP-hard, there is no known optimal algorithm for BPP running in polynomial time. However, Garey and Johnson [1] cite simple heuristics which can be shown to be no worse (but also no better) than a rather small multiplying factor above the optimal number of bins. The idea is straightforward: starting with one empty bin, take the objects one by one and for each of them first search the bins used so far for space large enough to accommodate it. If such a bin can be found, put the object there, if not, request a new bin. Putting the object into the first available bin found yields the First Fit (FF) heuristic. Searching for the most filled bin still having enough space to accommodate the object yields the Best Fit (BF), a seemingly better heuristic, which can, however, be shown to perform as well (as bad) as the FF, while being slower. Other possible approaches for tackling the BBP are described in [2] and [3].

1.2 Methodology Development

A great majority of natural and artificial systems are of complex nature and scientists quite often choose to work on systems simplified to a minimum number of components in order to observe "pure" effects. An alternative approach, often known as the *complex systems dynamics* approach [4], is to simplify as much as possible the components of the system, so as to take into account their large number.

Complex dynamical systems in general show interesting and desirable behaviour: *flexibility*, *robustness*, and *massively parallel* mode of operation. Systems of this kind abound in nature. Many of the dynamical systems used for computation find their equivalent in nature. Examples include genetic algorithms [5], spin glass models [6], connectionist architectures [7], reaction-diffusion systems [8], simulated annealing [9], and ant colony models [10].

An important notion in dynamical computational systems is that of *interaction*. The interaction could be based on *cooperation* or *competition*. In cooperation a collection of agents interact by communicating information, or hints (usually concerning regions to avoid or likely to contain solutions) to each other while solving a problem. The information exchanged may be incorrect at times and should alter the behaviour of the agents receiving it. An example of co-operative problem solving is the use of the *genetic algorithm* to find states of high fitness in some space. In a genetic algorithm members of a population of states exchange pieces of themselves or mutate to create a new population, possibly containing states of higher fitness. Another example is the *ant colony search model* in which a trail laid by previous ants significantly changes the behaviour of other ants. In competition agents are "fighting" for common resources. In general it is possible for natural alliances to emerge spontaneously within an overall competitive environment depending on the type and/or function of the individual agents. An example of a competitive search model is presented in section 2.

1. Load each object into an empty box. Initialize the strength attribute of each box to be equal to its empty space, i.e., *capacity – object's weight*.
2. Select randomly two boxes and initiate an inter box operation (IBO). IBO is an exchange of objects between two boxes (ref. fig. 2).
   a) If IBO is successful (i.e., at least one object exchange has occurred) then update *strength* and the list of *contained objects* of both boxes. If one of the boxes is empty, then delete it from the set of agents. Otherwise, reset the successive *interaction failures* attribute of the stronger box (i.e. the box with larger strength attribute) to zero.
   b) If IBO fails (i.e., no objects exchange has occurred), then increment the successive *interaction failures* attribute of the stronger box. If it exceeds an interaction failure threshold, then decrement the *strength* attribute of the stronger box by a predefined constant ε.
3. If termination criteria are not satisfied, then goto **2.** Otherwise exit with the current state of box agents as the final solution.

Fig. 1. The Many-agent Search Model. The implemented termination criterion is the number of IBO. ε is defined as percentage of the bin's capacity.

1. Generate all k -tuples of objects in each box, for some $k=1$ to C (fight complexity constant) as shown in fig. 3
2. Add the empty space of the *weak* box to its k-tuples as shown in fig. 4, where the empty space is determined by:
$$empty\_space = capacity - \sum_i weight_i$$
3. If the two boxes can exchange objects then find the best substitution (gain), i.e. an exchange of two k-tuples after which the empty space of the *strong* box increases at most (see fig. 5).

Fig. 2. The IBO algorithm

In this paper a dynamics approach is used to tackle an ordering problem — loading of objects into minimal number of boxes. Three different algorithms are used for solving the problem — a many-agent search model (MA), a genetic algorithm (GA), and an ant colony search model (AC). A common feature of the three algorithms is that they are complex dynamical systems used to do computation. Initially the systems are brought into an initial state corresponding to particular instance(s) of the problem to be solved. Then they are allowed to evolve according to their own dynamics. The final state of this evolution is taken as a solution of the problem. The computation to be performed is contained in the dynamics of the systems which are determined by the nature of the local interactions between many simple elements. Complex dynamical systems usually exhibit emergent properties and the behaviour of the system as a whole can no longer be viewed as a simple superposition of the individual behaviours of its elements, but rather as a side effect of their collective behaviour. The emergent property of the presented systems is the optimization of the cost function of the loading problem, i.e. the number of bins in the solution.

In section 2, a many-agent search model based on competition is described and computational complexity issues are discussed. In section 3, a genetic algorithm is presented and the crucial issues of representation and appropriate operators are discussed in detail. In section 4, the ant colony search model is defined and some generalizations are described. Experimental results are presented in section 5.

2. Many-agent Search Model Based on Competition

The many-agent system consists of simple agents possessing only limited knowledge of how to interact with other agents. The agents in the system are called *boxes* and each box has several attributes: *capacity*, *strength*, number of successive *interaction failures*, and a list of *contained objects*. The loading problem considered here is one dimensional and has only one constraint — box *capacity*.

The inter box operation (IBO) is a local interaction between two boxes. It is used to implement the competitive drive in the evolution of the system. One of the boxes is referred to as *strong*, and the other is referred to as *weak*

Fig. 3. k-tuples of objects

Fig. 4. k-tuples with added empty space

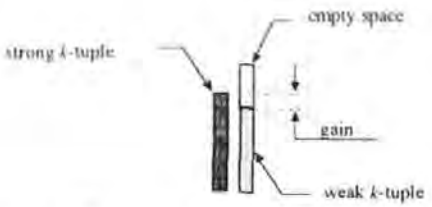


Fig. 5. Gain after an exchange of two k-tuples The objects to be packed are characterized by one attribute, called *weight*. Input to the algorithm is a set of objects to be loaded and box capacity. The MA algorithm is shown in fig 1.

(determined by the value of the strength attribute). The IBO algorithm is shown in fig. 2

The complexity of IBO is: $O\left(\binom{K}{C}\right)$, where $K=max(m,n)$, $m=$*number of objects in the first box, and* $n=$*number of objects in the second box*. In the course of its evolution the MA search model reduces the number of agents (bins). The possibility that the *strength* attribute may differ from the empty space (ref. fig. 1, step 2b) is introduced as a mechanism for escaping local optima. Thus a strong box which could not complete a successful IBO (i.e. could not gain empty space) becomes weaker and eventually other boxes try to fill it in.

3. Genetic Algorithm

3.1. Representation

The genetic algorithm presented in this section has fixed-length order-based representation coupled with a First Fit (FF) decoder [11] (see fig. 6).

Let $Objs=[o_1,\ldots,o_n]$ be the set of objects to be loaded. Each permutation of these objects is a *valid* chromosome. The number of different chromosomes (the representation space for the problem) is $n!$. A very important issue to be discussed here is that the proposed representation does not comply with the minimal redundancy principle [12] (i.e. each member of the solution space should be represented by as few as possible distinct chromosomes, in order to reduce the size of the actual search space). This is compensated by special operators which work on the solution space rather than the representation space.

A straight forward observation is that if the objects in the chromosome are sorted in decreasing order of their weight, then after applying the FF mapping the solution of the FFD heuristic is obtained. Thus the FFD heuristic (which guarantees that $\mathrm{FFD}(Objs)\le\frac{11}{9}\cdot\mathrm{OPT}(Objs)+3$ [13]) can be easily integrated into the initial population of the GA.

A salient feature of the selected representation and mapping is the context sensitivity of the genes in the chromosome, i.e. the probability that object o is in a particular bin depends on the objects to the left of o in the chromosome and does not depend on the objects to the right. This observation could be useful in designing genetic operators. When the intention is to preserve certain schemata from a parent chromosome they should be placed at the left side of the child chromosome.

3.2 Operators

Theory [14] and evidence suggest that search algorithms perform better when augmented with domain-specific knowledge. This is the reason for designing problem-specific operators that work both in the representation and solution spaces and utilize knowledge of how the mapping process works.

Uniform scramble sublist (USS): Randomly (with probability P_s) select objects from the chromosome and permute them. P_s controls the destructiveness of the operator.

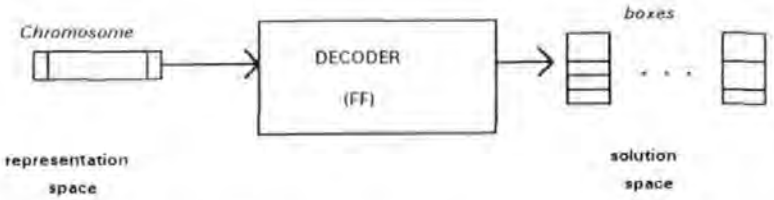


Fig. 6: Problem representation. The First Fit decoder is defined as follows: starting with an empty bin, take objects one by one from left to right and for each of them try to put it into the first available bin. If no such bin is found, request a new bin

Reduced hypercube (RH): The parent chromosome is randomly split; one child is constructed by appending the left part of the parent chromosome and a random permutation of the right part. Using the same split, other children are constructed in the same way. The number of children is a parameter of the operator. The assumption behind the design of this operator is that the first part of the parent chromosome may be considered as a good partial solution which is preserved by the FF mapping with high probability. Then by creating several children exploiting that particular partial solution, we construct a hypercube around it.

Reduced hypercube 2 (RH2): Let S be the solution of the parent chromosome, and S' be the solution S sorted in ascending order of the bins' empty space. To form a child chromosome get the objects from the first bin in S' and place them at the beginning (left) of the child chromosome. Then get the objects from the second bin in S' and append them to the end of the just formed partial child chromosome. Repeat the process with the rest of the bins. Now on the new child chromosome apply the reduced hypercube operator. This operator favours more filled bins and preserves them in the child chromosome with above average probability.

3.3 Fitness Function

As the goal of the problem is to minimize the number of bins it is quite natural for our fitness function to attribute more credit to solutions with fewer bins. We are also looking for a fitness function that is able to differentiate between two solutions having the same number of bins. One useful measure could be the variance of bins' empty space in a particular solution: greater the variance—better the solution. This measure is also justified in applications where only partial solutions are required at a time and the problem is iterated, i.e. new objects come on a continuous basis. The fitness function is defined as follows:

$$f = n + \frac{\alpha}{\text{variance}_{ES}}$$

where n is the number of bins in the solution, variance_{ES} is the variance of the empty space in the solution, and α is normally defined to give total preference to solutions with less number of bins.

3.4 Genetic Algorithm Implementation

The following GA scheme is implemented: (a) Initialize a population of M chromosomes. (b) Evaluate each chromosome in the population. (c) Create N new chromosomes using the defined genetic operators, (d) Evaluate the new chromosomes and place them in the population, (e) Select the M most fitted individuals and goto (c) until time is up.

A population size of $M=70$ is adopted. The selection procedure uses roulette wheel selection and linear fitness scaling (normalization). The probabilities of the operators are also implemented as a roulette wheel selection. The operators weights are: USS-30, RH-15, and RH2-30.

4. Ant Colony Search Model

Problems like the travelling salesman problem (TSP) and bin packing can be represented as a sequence of n items (n cities to be visited or n objects to be packed), where the actual order of the sequence determines a particular solution to the problem. Thus, in general, the search space consists of all $n!$ permutations.

The ant cycle algorithm is first proposed in [15] and is defined as follows: The problem is represented as a connected graph the nodes of which are the n items. The edges are connections from one item to another and represent a data structure that stores the connectivity information in terms of the trail τ_{ij} left by the ants in the course of the algorithm's execution. The trail is defined by:

$$\tau_{ij}(t+n) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t, t+n)$$

where ρ is an evaporation constant ($0<\rho<1$), t is the time at the beginning of a tour, and $t+n$ is the time at the beginning of the next tour (a tour is a permutation of the n items). $\Delta\tau_{ij}(t, t+n)$ is defined as follows:

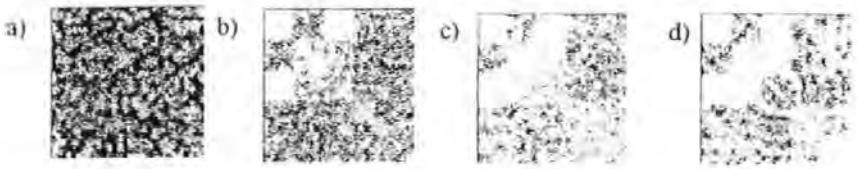


Fig: 7. Evolution of the connectivity pattern during different stages of the ant colony run: a) is the initial (random) pattern and d) is the pattern when the ant colony has converged. b) and c) represent intermediate patterns.

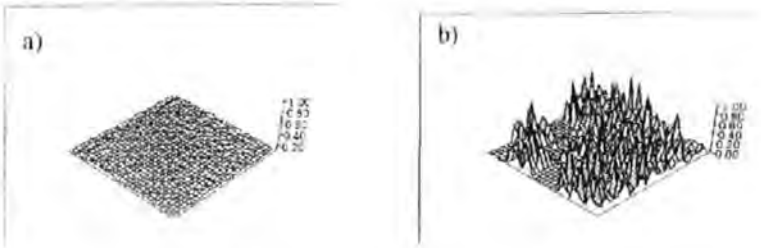


Fig. 8. Trail left by the ants: a) initial (random) trail at the beginning of the run; b) trail at the end of the run (i.e. when the ants' search has converged).

$$\Delta\tau_{ij}(t,t+n)=\sum_{k=1}^{m}\Delta\tau_{ij}^{k}(t,t+n)$$

where m is the number of ants. $\Delta\tau_{ij}^{k}(t,t+n)=Q(f_k)$ if edge(i,j) is in the tour of ant k, and is zero otherwise. In general $Q(f_k)$ is proportional to the fitness f_k of ant k. The same fitness as in the GA is adopted in this work.

Then during the next tour the probability to visit item j when being at item i is:

$$P_{ij}(t)=\begin{cases}\dfrac{[\tau_{ij}(t)]^{\alpha}\cdot[\eta_{ij}(t)]^{\beta}}{\sum_{k\in allowed}[\tau_{ik}(t)]^{\alpha}\cdot[\eta_{ik}(t)]^{\beta}} & \text{if } j\in allowed\\ 0 & \text{otherwise}\end{cases}$$

where *allowed* is the set of items not visited for that particular tour and η_{ij} is a local heuristics. α and β define a trade-off between the local heuristics and the ant colony search. If the local heuristics η_{ij} could not be defined then $\eta_{ij}=1$ is assumed for all ij.

The algorithm runs as follows: Initially P_{ij}'s are randomly initialised for some small values (fig. 7a, fig. 8a) and m ants are allowed to make a tour. Then the solutions are compared and trail is laid on the edges comprising the tours proportionally to the ants' fitness (fig. 7b,c,d, fig. 8b). This alters the P_{ij} values so that on the next tour the probability of repeating (part of) previous good tours increases. This is reminiscent of *schemata propagation* in genetic algorithms where building blocks of high fitness pass from one agent to its offspring.

4.1 Underlying Theory

In order to theoretically investigate the ant colony search there is a need for an underlying theory. In this subsection we outline the similarities between ant colony search and epidemic processes.

Epidemic processes are common in nature. Besides the spread of diseases they also characterise such diverse processes as fire spread, starfish outbreaks, invasions of exotic plants, etc. Epidemics can be characterised as *invasion percolation* [16]. Invasion percolation refers to flows that create their own channels through a medium. As with all percolation processes epidemics display critical behaviour. That is for some parameter ν associated with the process, the process exhibits a "phase change" from non-spread to spread, at some *critical value* ν_C. In this respect epidemics resemble a large class of phenomena, ranging from collapsing sand hills to nuclear chain reactions [17]. For the ant colony search, the critical parameter is the trail propagation rate ν which is the probability that one ant would influence its behaviour due to trail sensing. When $\nu < \nu_C$ (i.e. the epidemic dies out naturally) the trail evaporates so quickly that P_{ij} virtually does not depend on τ_{ij}. Therefore, the individual ants proceed the search selfishly with their own search

Problem Size	30	60	90	120	150
optimal solution	9	18	27	36	45
FFD	11	22	33	44	55
MA	9.1	18.4	27.6	36.3	45.3
GA	9	18	27	36	45
EAC	9	18	27	36	45

Table 1. Experiments with FFD worst case distribution. For this particular distribution of object weights all our approaches outperform the FFD heuristic. Therefore, including the FFD solution in a hybrid model utilising (all) our approaches is justified.

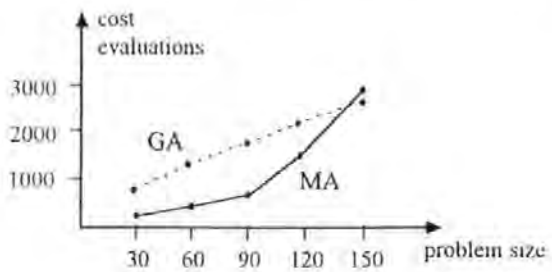


Fig. 9. Number of fitness evaluations as a function of the problem size

strategies. When $v > v_c$ (i.e. the epidemic spreads indefinitely) the trail accumulates indefinitely and due to diffusion spreads uniformly on the entire neighbourhood, i.e. the trail distribution exhibits maximum entropy and could not serve as a guide to the ants. In this case P_{ij} again favours the individual search strategies. (Please note that in both cases $\eta_{ij} = 1$ would imply random search).

4.2 Extensions of the Ant Colony Metaphor

We begin this subsection with some initial definitions of distributed co-operative search [18]. Co-operative search methods are based on modifying individual search methods. A useful distinction is whether a method is *complete* or *incomplete*. Complete methods systematically examine states and are guaranteed to either eventually find a solution or terminate when no solution exists. By contrast, incomplete methods explore more opportunistically and may miss some states in the search space, hence they can never guarantee a solution does not exist. For parallel searches, a further issue is whether to split the search space among the agents. In the simplest case, each agent examines the entire search space. However this can mean a single state is examined by more than one agent during the search. This can be avoided by partitioning the search space into disjoint parts and assigning one to each agent. In the partitioned search, agents only examine states in their assigned part of the space thus avoiding unnecessary duplicate examination of states. Restricting each agent to examine a state at most once, as well as partitioning the search space so that a state is not examined by more than one agent, may improve performance somewhat, but far less than the enhancement achieved by co-operation [18][19][20].

Now the ant colony search model is generalized to include search agents with different strategies. The completeness of the overall search depends in general on the completeness and the complementarity of the individual strategies. The search space is not explicitly split among the agents. However, the individual strategies are implicitly competing for a common resource — CPU time. The fitness measure of the competition is the success rate of each strategy measured from the beginning of the evolution. The overall evolution process resembles parallel competitive hypothesis testing from evolving statistical sample (the current population). The hypothesis are the prior assumptions within which each individual search strategy has been designed to be effective. For example, consider two search strategies: a_1 and a_2 designed to be effective when applied to distributions of problem instances h_1 and h_2 respectively. When applying the ant colony to an instance of distribution h (for which we know that h may be either close to h_1 or h_2) then the generated search process could be viewed as hypothesis testing. The assumption is that if the current problem instance is generated from h_1 then a_1 will take control over a_2 (please note that a_1 was

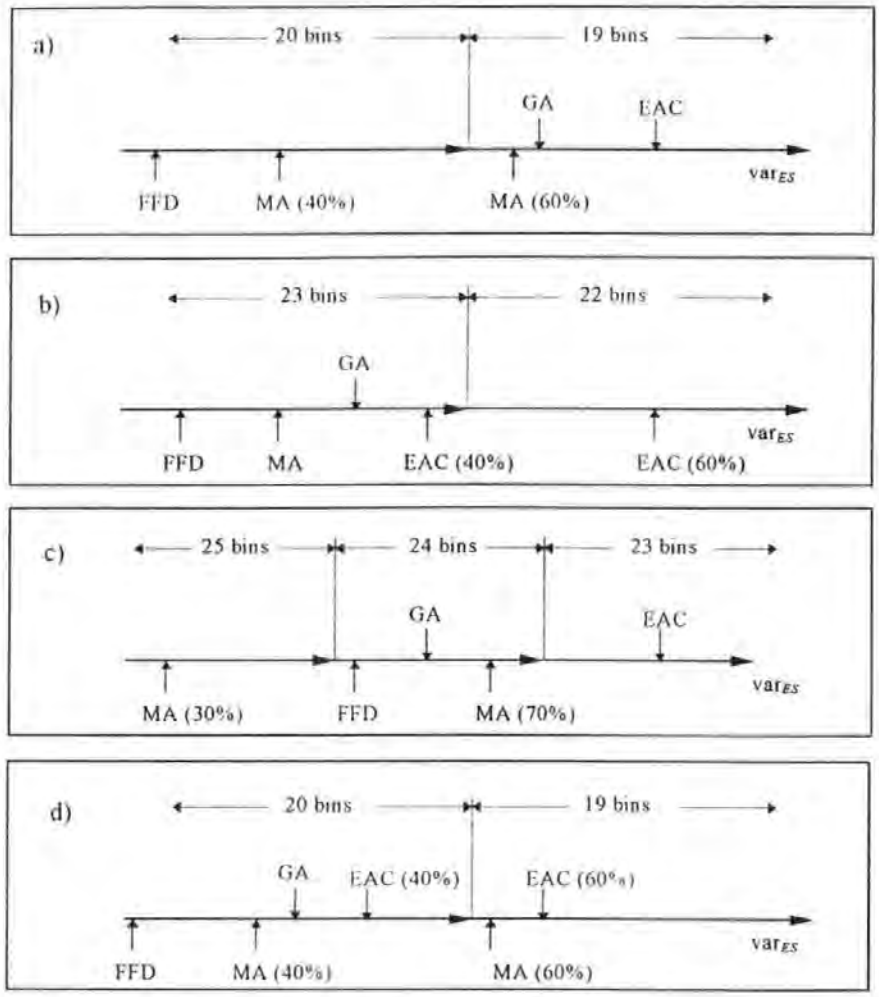


Fig. 10. Experiments with uniform distribution of objects. The hybrid search model (EAC) outperforms any of its comprising individual search strategies (FFD, GA, and MA). Therefore, individual performance diversity and complementarity are of crucial significance to the performance of the hybrid search model.

initially designed to be effective on h_1). This is somewhat analogous to the self-adaptation notion in evolutionary programming [21] and genetic algorithms [22].

For the BPP the extended ant colony (EAC) works as follows: Two new types of search ants are allowed — m agents using the MA strategy and k agents utilising the GA strategy. At each generation of the GA trail is laid on the tours of the k best solutions. (To keep the grouping effect of the Bin Packing solutions trail is laid on all the edges connecting each of the objects from a particular bin.) Then m ants are allowed to make a tour and the MA search is applied on each tour. Trail is laid (and superimposed) proportionally to the fitness of each tour. Therefore, in the current implementation of the EAC search model both the GA and the MA strategy compete against each other by changing the ants' behavior through the trail value.

5. Experiments

Initially a distribution of object weights which has been proved to be the worst case for the FFD algorithm is used. Then experiments with randomly distributed object weights are considered.

5.1 FFD Worst Case Distribution of Object Weights

The distribution is defined as follows: $m = 1,2,3,\ldots, \varepsilon = 0.01$; $weight(obj_i) = \frac{1}{2} + \varepsilon$ (for

$1 \le i \le 6m$). $weight(obj_i) = \frac{1}{3} + 2\varepsilon$ (for $6m \le i \le 12m$), $weight(obj_i) = \frac{1}{4} + \varepsilon$ (for $12m \le i \le 18m$), $weight(obj_i) = \frac{1}{4} - 2\varepsilon$ (for $18m \le i \le 30m$).

Experiments are done for values of m from 1 to 5, i.e. for problem sizes ranging from 30 to 150 objects. Empirical results are summarized in table 1. Fig 9 shows the number of the required fitness evaluation calls as a function of the problem size. Results are averaged over ten independent runs.

Table 1 shows that all our algorithms (MA, GA, and EAC) outperform the FFD heuristic for this particular distribution of object weights. Empirical results also reveal that the performance of our techniques seems to be somewhat "complementary" to FFD and therefore, a hybrid search model, (possibly) including all techniques, is justified.

5.2 Uniform Distribution of Object Weights

Four uniformly random sets of objects are generated using the following parameters: *min. value: 0.05, max. value: 0.65, resolution: 300, and problem size: 50.* Four algorithms are empirically compared: FFD, MA, GA and EAC. Results are averaged over 10 independent runs and shown in fig. 10.

In fig. 10 each graph is divided by vertical lines into sections. Each section corresponds to a particular number of bins in the solution. Better solutions are placed to the right of the graph. Within each section solutions are ranked according to the variance of the empty space. Again better solutions are placed to the right of the graph. When a method produces results with different number of bins it is shown more than once in the corresponding sections of the graph.

In this experiment, the solution of the FFD heuristic is included in the initial population of the GA as shown in section 3.1. The hybrid search model (EAC) generally outperforms any of its comprising individual search strategies. Therefore, their performance diversity is very important for the overall performance of the hybrid model (fig. 10).

6. Conclusions

The objective of this paper was to develop, apply, and empirically compare using a well-known NP-complete problem, various evolutionary metaphors. An attempt is made to define in a unified framework (through the generalized ant colony search model) the hybrid approach, i.e. the mutual co-operation of well-defined heuristics solving a problem together. Empirical results (fig. 10) confirm the potential power of such an approach. It is believed that the evolution and self-adaptation of *a priori* assumptions about the problem (instance) at hand are the only legitimate tools to overcome the practical implications of the limitation theorems on search [23].

Current research involves the generalized ant colony search model applied to continuous function optimization [20]. Open questions for further research are to address the stability issues when designing "multi-species" distributed co-operative searches, balance between co-operation and competition, and increasing the complexity of the problems.

References

[1] Garey M. and Johnson D., Computers and Intractability - A Guide to the Theory of NP-completeness, W.H.Freeman Co., San Francisco, USA, 1979

[2] Martello S. and Toth P., Bin Packing Problem, Chapter 8 in Knapsack Problems, Algorithms and Computer implementations, John Wiley and Sons Ltd., England, 1990

[3] Falkenauer E., New Representation and Operators for Genetic Algorithms Applied to Grouping Problems, in *Evolutionary Computation*, Vol.2 , No. 2, 1994, pp. 123-144

[4] Weisbuch G., Complex Systems Dynamics, Lecture Notes Volume II, Santa Fe Institute, Studies in the Sciences of Complexity, Addison-Wesley, 1991

[5] Holland J. H., Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications in Biology, Control, and Artificial Intelligence, University of Michigan Press, Ann Arbor, 1975

[6] Kinzel W., Learning and Pattern Recognition in Spin Glass Models, 1985

[7] Rumelhart D.E., Hinton G.E., and Williams R.J., , Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1, The MIT Press, Cambridge, MA, 1986

[8] Steels L., Artificial Intelligence and Complex Systems, AI-MEMO 88-2, AI-lab, VUB, Brussels, 1988

[9] Kirkpatrick S., Gelatt C.D., and Vechi M. P., Optimisation by Simulated Annealing, *Science*, Vol. 220, No. 4598, May, 1983

[10] Dorigo M., Optimization, Learning, and Natural Algorithms, *PhD Thesis*, Politecnico di Milano, ITALY, 1992 (in Italian)

[11] Bilchev G., Evolutionary Algorithms for the Bin Packing Problem, *MSc Thesis*, New Bulgarian University, 1994 (in Bulgarian)

[12] Radcliffe N., Forma Analysis and Random Respectful Recombination, in *Procs. of the Fourth ICGA, San Diego*, 1991

[13] Baker B.S., A new proof for the first fit decreasing bin-packing algorithm, *J. Algorithms* 6, 1985, pp. 49-70

[14] Wolpert D. and Macready W., No Free Lunch Theorems for Search, SFI-TR-95-02-010, 1995

[15] Colorni A., Dorigo M., and Maniezzo V., Distributed Optimization by Ant Colonies, in Procs. First European Conference on Artificial Life, Varela F., and Bourgine P. (eds.), Paris, Elsevier, 1991, pp. 134-142.

[16] Wilkinson D. and Willemsen J.F., Invasion percolation: a new form of percolation theory, *Journal of Physics A*, 1988 (16), pp. 3365-3376

[17] Bak P. and Chen K., Self-organized criticality, *Scientific American*, 1991 (265), pp. 26-33

[18] Clearwater S., Huberman B., and Hogg T., Cooperative Problem Solving, in B. Huberman, editor, *Computation: The Micro and the Macro View*, pp. 33-70, World Scientific, Singapore, 1992

[19] Bilchev G. and Parmee I.C., The Ant Colony Metaphor for searching Continuous Design Spaces, in *LNCS 993: Evolutionary Computing 2*, edited by T. Fogarty, Springer-Verlag 1995, pp. 25-39

[20] Bilchev G. and Parmee I.C., Constrained Optimisation with an Ant Colony Search Model, accepted at *Adaptive Computing in Engineering Design and Control '96*, University of Plymouth, UK, 1996

[21] Fogel L., Angeline P., and Fogel D., An Evolutionary Programming Approach to Self-Adaptation on Finite State Machines, in Evolutionary Programming IV: *Procs. of the Fourth Annual Conference on Evolutionary Programming*, Cambridge, MA, The MIT Press, 1995

[22] Back T., Self-Adaptation in Genetic Algorithms, in Varela F. and Bourgine P (eds.), *Procs. of the First European Conference on Artificial Life*, Cambridge, MA, MIT Press, 1992, pp. 263-271

[23] Radcliffe N. and Surry P., Fundamental Limitation Theorems on Search Algorithms: Evolutionary Computing in Perspective, *LNCS 1000*, Springer-Verlag, 1995

Constraint Handling for the Fault Coverage Code Generation Problem: An Inductive Evolutionary Approach

George Bilchev* and Ian Parmee*

*Plymouth Engineering Design Centre, University of Plymouth, PL4 8AA, UK
GBilchev@plymouth.ac.uk

Abstract. Real world problems quite often are constrained and their successful solution requires the application of an appropriate constraint handling technique. The lack of a uniform methodology for handling nonfeasible points largely predetermines the current best practice – the investigation of some problem-specific operators which search (within) the feasibility boundary in an efficient way. In this paper we apply this approach to a real world problem provided by Rolls Royce and Associates Ltd., and show how to design feasibility preserving operators that map the feasibility region onto itself. Some of our results provoke new ideas of how to modify real-time test and monitoring systems so as to increase their reliability.

1. Introduction

Various constraint handling techniques have recently emerged [1]. However, there is still no uniform methodology for handling unfeasible points. Current best practice involves the investigation of some problem-specific operators, which search the feasibility boundary in an efficient way [2]. This idea is based on the seemingly reasonable assumption that in real world problems the constraints and the objective functions are conflicting and therefore, the constraint global solution lies on the boundary of the feasible region.

In this paper we use a real world problem (section 2) to demonstrate a very efficient constraint handling technique. It consists of defining the feasible region in terms of the independent variables and designing feasibility preserving operators (i.e. operators that map the feasible region onto itself) (section 3). The existence of such a closed form description of the feasible space leads to a minimal redundancy problem representation [3] and could significantly reduce the search space. Currently the feasibility preserving constraint handling technique is being applied successfully to the optimization of real valued functions and linear constraints [4], and for combinatorial problems. In this paper we extend the applicability of the approach to include the fault coverage test code generation problem [5] with additional constraint requirements imposed by the designers of the circuit's logic.

We consider the reduction of a search space to be one of the most efficient approaches for solving any search problem. This idea has been fundamental for

many of the existing search methodologies, including branch-and-bound, clustering, etc. In this paper we also propose to integrate the search space reduction approach with an evolutionary search engine (section 4). The idea has already produced successful results when applied to optimization of real valued continuous functions [6][7].

2. The Fault Coverage Test Code Generation Problem

Test and Monitoring Systems (TAMS) are widely used for real-time testing of the functionality of electronic circuits (fig. 1). Basically they operate by regularly initiating a test cycle on the circuit and monitoring the fault status. An integral part of the TAMS is a fault coverage test code consisting of a set of input test vectors and a set of expected output vectors. New circuits cannot be used until the fault coverage code is updated with a new set of comprehensive test vectors (fig. 2).

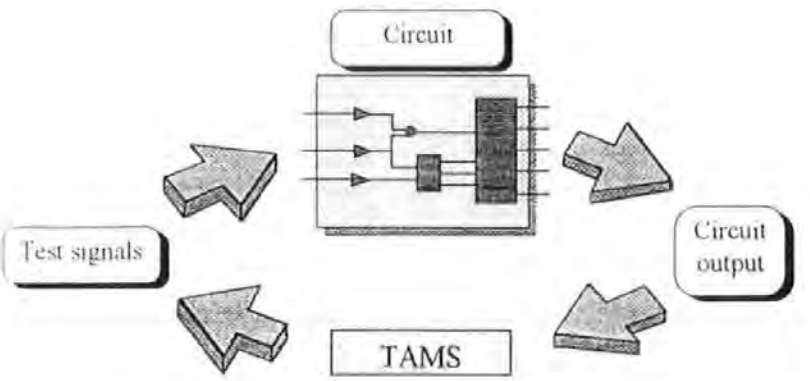


Fig. 1. Overview of test and monitoring system (TAMS)

Fault analysis is the process used to determine the fault detection coverage of a particular design. The fault analysis process for a design involves the optimization of the input stimulus to fully exercise all components to increase the testability, while logic simulation involves the optimization of the functionality of the design. These tasks are very different processes, and both tasks are necessary within the design development process. The fault analysis process fits within the product development cycle after the initial functional verification of the design and before the physical hardware testing of the product.

The amount of fault coverage within a design depends on the following two factors: (1) comprehensiveness of the test code, and (2) inherent testability of the logic design. In this paper we concentrate on the first factor and formulate the problem of finding an effective set of input test vectors as a *search problem*.

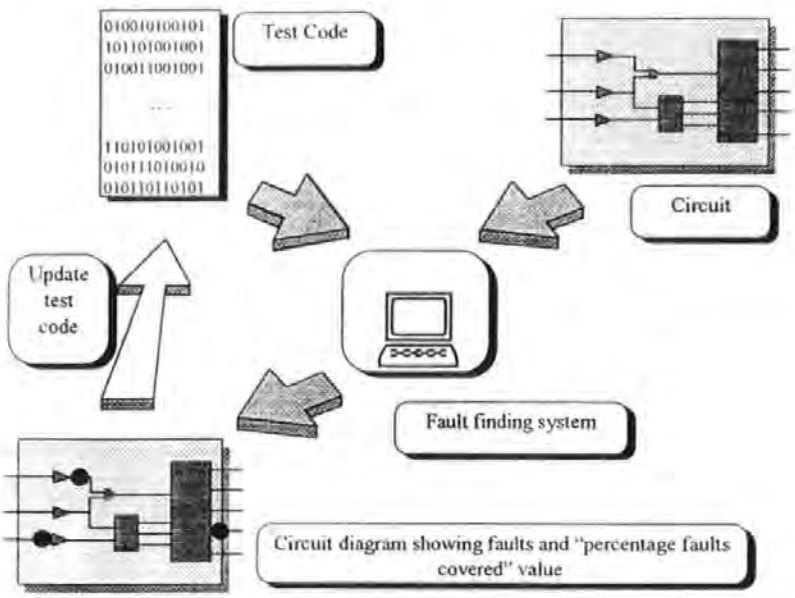


Fig.2. The process of finding the most efficient fault coverage test code: The circuit is modeled and faults are simulated. Using information from the fault analysis the task is to design the most comprehensive test vectors (the white arrow).

3. Handling Constraints

Usually there are various constraints imposed on the test codes. For example, the size of the test code may be constrained by hardware requirements of the test and monitoring system. There may also be a number of constraints concerning the possible combinations of input signals. The task is to automate the process of finding the most effective test code, i.e. the code maximizing the fault coverage (fig. 2).

The requirement that the number of test vectors must be exactly N is represented directly by the coding scheme of the problem. A sample from the associated fitness landscape of the search problem would consist of N vectors each of length m bits.

The second type of constraints impose strict requirements on the possible combinations of values within each individual test vector. The designers of the circuit define the set of legal combinations in terms of the legal states of a number of channels (fig. 3). Each channel is a logical grouping of input bits (for example, bits No. 2, 5, and 7 could form logical channel 1). Collectively the legal states of all channels define a set of legal (*supporting*) templates of the form:

```
1*0**1011***
```

where * is a don't care symbol. Each template could be viewed as a generator of a particular fraction (subspace) of the original search space. Therefore, the set of all legal templates defines the feasible region. The existence of a such closed form description of the feasible region greatly influences the selection of a constraint handling technique. In our case, it seems appropriate to maintain a population of legal samples by designing feasibility preserving search operators.

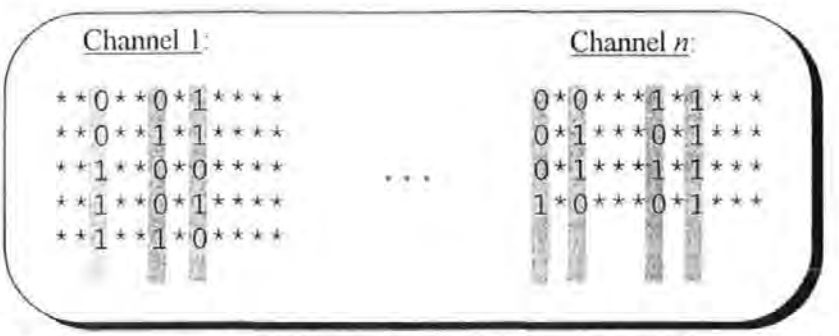


Fig. 3. A set of legal test codes is defined by the legal states of a number of logical channels.

When applied to a feasible point(s), a feasibility preserving operator always produces another feasible point(s). For the test code generation problem we have designed two versions of mutation and one of crossover which comply with the selected constraint handling technique.

mutation 1: (*i*) find the supporting template of the parent chromosome, and (*ii*) apply uniform mutation to the values of the don't care bits.

```
100101011011    parent chromosome
1*0**1011***    parent's supporting template
110111011001    offspring
```

mutation 2: (*i*) find the supporting template of the parent chromosome, (*ii*) change it randomly by another supporting template while keeping the values of the don't care symbols.

```
100101011011    parent chromosome
1*0**1011***    parent's supporting template
0*1**0011***    new supporting template
001100011011    offspring
```

`crossover`: (*i*) find the supporting template of both parents and (*ii*) apply uniform crossover to the don't care bits.

parents	1[redacted]01[redacted]1011011	0[redacted]1101001100[redacted]1
templates	1*0**1011***	0*1**0011***
offspring 1	11011011001	0[redacted]10[redacted]00110[redacted]1

5. The Inductive Genetic Algorithm

In general the inductive approach generates a solution step by step, beginning from the so called *base* of the induction and at each step following an *induction rule* to update (i.e. induce) the solution. In mathematics induction is a rigorous proof technique while in the context of adaptive search it is used to approximately induce a solution to a particular problem. Previous research [6][7] well justifies the potential power of the inductive approach in the context of search.

Applying the inductive approach to the fault coverage code generation problem requires a slight reformuation of the problem. The original problem is:

> Given a number *N* (the maximum number of fault coverage test vectors) find a sequence of *N* test vectors that maximizes the fault coverage.

It can be easily reformulated as:

> For each *k* = 1 to *N* find a sequence of *k* test vectors that maximizes the fault coverage.

In this case the inductive formulation also gives meaning to intermediate solutions. Suppose for example that for some k, $1<k<N$, we know a sequence of test vectors which gives *satisfactory* fault coverage. The term satisfactory means that with k test vectors we couldn't expect to cover more faults than those already discovered by the sequence. This is a relative judgement regarding the particular circuit and can serve as an efficient stopping condition instead of the usual maximum number of generations (refer to step 6 of the algorithm). Now if for k test vectors we have already achieved a satisfactory level of fault coverage the task is to find a satisfactory fault coverage level for $k+1$ test vectors. The main power in the inductive approach is the assumption that the satisfactory fault coverage level for $k+1$ test vectors could be derived from the satisfactory level of the k test vectors. If this assumption is true then it produces an efficient search engine with computational complexity determined only by the computational complexity of the inductive step.

The Inductive Genetic Algorithm (IGA) combines the evolutionary search engine with an inductive fitness function. The overall structure of the IGA is as follows:

1. Initialize a partial solution for $N = 1$ (i.e. a sequence of one test vector only)

100101011011

2. For $k = 2$ to N do (search for the best kth vector that complements the already existing partial solution)

3. Initialize a population of test vectors

4. Append each test vector to the partial solution, evaluate it and assign fitness

5. Reproduce according to the fitness obtained in 4. Typical operators include versions of mutation and crossover. Update the population.

6. If not end of generation, goto 4

7. Update the partial solution, increment k, goto 2

Steps 3, 4, 5, and 6 constitute a genetic algorithm. Steps 1, 2 and 7 implement the inductive approach. The overall algorithm could also be viewed as a genetic algorithm with dynamic fitness function, i.e. the fitness function changes at each generation.

6. Experiments

The associated fitness function with our model is $f = \frac{n}{N} \cdot 100\%$, where n is the number of covered faults and N is the number of all faults in the fault population.

Fig. 4 shows the performance of the Inductive Genetic Algorithm (IGA) on a fault coverage problem consisting of 200 possible faults. The number of input test vectors is 24. The search effort at each inductive step controls the trade-off between the computational complexity and the expected quality of results. The family of all possible trade-off points define the *performance trade-off* front. It is a measure of the expected gain of the quality of results as a function of the computational expense.

The fault coverage achieved for 9,000 fitness function calls is 71.5%. Previous experiments done at Rolls Royce and Associates Ltd. involved a classical genetic algorithm in which the chromosome coded all the 24 input test vectors each of length 12 bits. The best produced fault coverage for 9,000 fitness function calls was 57%.

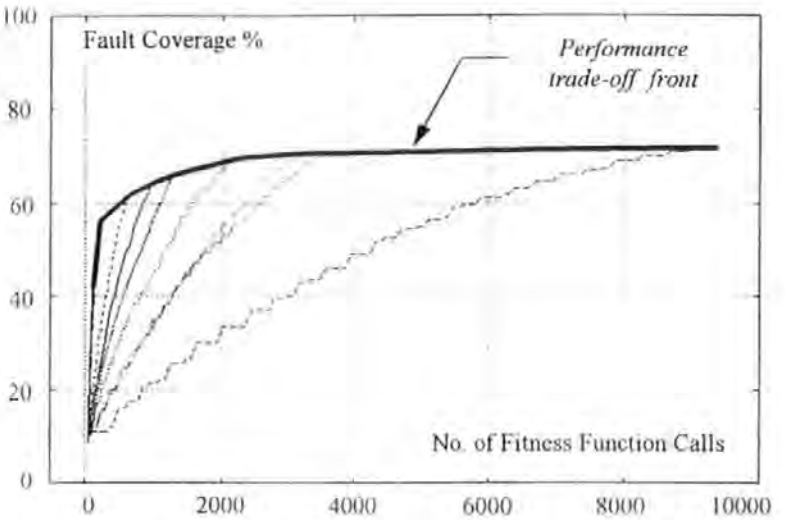


Fig 4. Runs of the Inductive Genetic Algorithm for eight different control parameter settings (population size, mutation rate, crossover rate, number of generations per inductive step). The family of all possible control parameter settings define the *performance trade-off front*, which is a measure of the trade-off between computational complexity and quality of results. The number of input test vectors is 24.

Fig. 5 shows the performance of the IGA as a function of the number of input test vectors. The same fault population of 200 faults as in the previous experiments is used. The IGA is able to find a set of 67 input test vectors that cover 100% of the fault population. If the hardware requirements allow the TAMS to use 67 test vectors then 100% fault coverage is achieved. However, in real world problems there are hard constraints imposed on the design task. For example, in our particular TAMS test code generation problem the number of input test vectors must be 24. Therefore, our objective is to design maximally comprehensive set of 24 test vectors (the design of the logic of the circuit is already fixed, so we regard the test coverage code generation problem as a search problem and do not address the inherent testability properties of the logic design). However, the availability of an algorithm that produces a population (of minimal size) covering 100% faults could be used to implement a test and monitoring system which uses several sets of 24 test vectors and applies them during different discrete time steps (interleaving).

Fig. 6 shows the effect of the constraints on the performance of the search engine. There are two graphs, each corresponding to a particular set of constraints. Each set of constraints is determined by a table of legal states as explained in section 4. Legal table 2 is derived from legal table 1 by reducing the number of legal states in channel 1. Therefore, legal table 2 corresponds to a more constrained instance of the fault coverage test code generation problem. Imposing constraints on the problem is equivalent to introducing *inherent untestability* in the circuit design. Legal identifying fault patterns could no longer be allowed and therefore, the corresponding faults would never be detected. As can be seen from fig. 6 this fact significantly influences the achieved fault coverage.

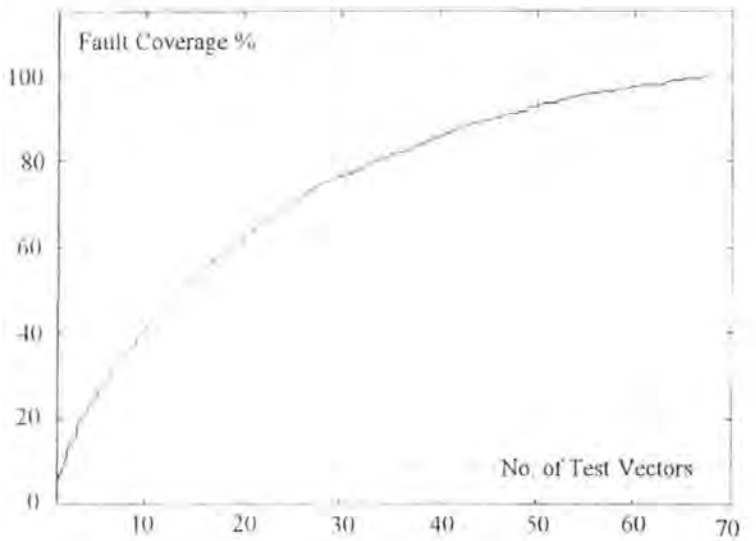


Fig. 5. The fault coverage as a function of the number of input test vectors. There are 200 faults in the fault population. An Inductive Genetic Algorithm is used to find the best set of test vectors. The number of fitness function calls in order to cover 100 percent of the fault population is approximately 10,000 and required sixty seven input test vectors.

7. Conclusions

In this paper we have described the development and application of an efficient constraint handling technique to a real world problem. As compared to previous experiments at Rolls Royce and Associates Ltd. our system has considerably improved the performance and the reliability of the test and monitoring system (TAMS). Also, some of our results suggest how to modify TAMS so as to increase real-time fault coverage.

Currently, there is no general way to find a closed form description of the feasible region. Our experience shows that it may prove to be worthy to spend some time

analysing the problem before resorting to the selection of the constraint handling technique. Several existing evolutionary search systems already comply with the above conclusions [2][4] and show improved performance as compared to previous constraint handling methods. Currently such systems are applicable only to particular classes of constraints (e.g. linear constraints) and future research may involve the extension of these classes.

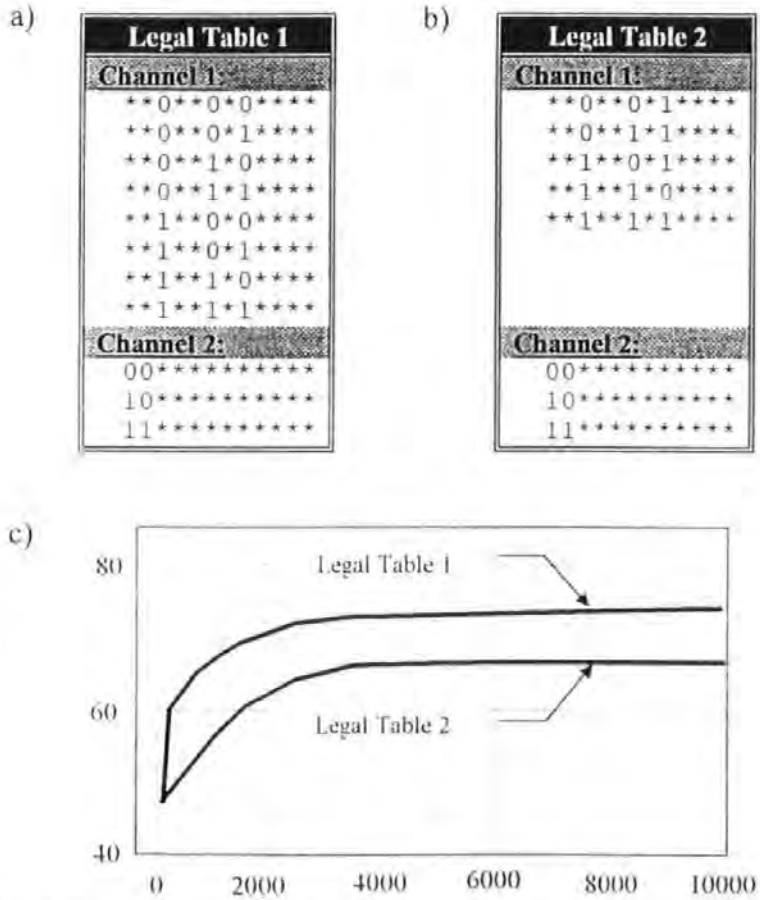


Fig. 6. Effects of constraints on the performance trade-off front of the IGA. The effect of imposing of constraints on the fault coverage problem (Legal Table 2) is equivalent to introducing inherent untestability in the circuit.

References

[1] Michalewicz Z., A Survey of Constraint Handling Techniques in Evolutionary Computation Methods, *The 4th Annual Conference on Evolutionary Programming' 95*, March 1-3, San Diego, USA

[2] Michalewicz Z., Nazhiyath, and Michalewicz M. A Note on Usefulness of Geometrical Crossover for Numerical Optimization problems, *the 5th Annual Conference on Evolutionary Programming' 96*, San Diego, USA

[3] Radcliffe N., Forma Analysis and Random Respectful Recombination, *Procs. of the 4th ICGA*, San Diego, 1991

[4] Michalewicz Z., Genetic Algorithms + data Structures = Evolutionary Programs, Springer-Verlag, 1992

[5] Bilchev, G., and Parmee I., The Inductive Genetic Algorithm with Applications to the Fault Coverage Test Code Generation Problem, *submitted to EUFIT'96*, Aachen, Germany

[6] G. Bilchev, and I. Parmee, "Learning the 'Next' Dimension", *Artificial Intelligence and Simulation of Behaviour'96*, Brighton, UK, April 1996

[7] G. Bilchev, and I. Parmee, "Inductive Search", *First International Contest on Evolutionary Optimization to be held during the 1996 IEEE Conference on Evolutionary Computation*, May 20-22, Nagoya, Japan, 1996

The Inductive Genetic Algorithm with Applications to the Fault Coverage Test Code Generation Problem

George Bilchev[1] Ian Parmee[1] Andrew Darlow[2]

[1]Plymouth Engineering Design Centre
University of Plymouth,
Plymouth PL4 8AA
UK
Tel : 01752 233508
Fax.: 01752 233505
E-mail:G.Bilchev@plymouth.ac.uk

[2]Industrial Systems Design Group
Rolls Royce and Associates Ltd
PO Box 31, Derby DE24 8BJ
UK
Tel 01332 661461 ext. 5694
Fax.: 01332 622950

1. Introduction

Recent growth of interest in algorithms inspired by natural processes has resulted in a number of real world applications. Techniques like genetic algorithms [1], evolutionary strategies [2], immune networks [3][4] and ant colony search models [5][6] are now widely accepted as robust general purpose search engines. However, recent advances in search theory [7][8] show that the only possible way to tackle the search problem is by incorporating *a priori* knowledge. Therefore, currently the evolutionary search framework is well suited for the preliminary engineering design stage, where models are coarse, there is a great deal of uncertainty and the objective is to grasp a general overview of the search landscape [9].

In this paper we present a novel evolutionary search engine which we believe has a great potential. We also present results regarding the application of our approach to a real world engineering design problem, namely, the design of efficient fault finding test code at Rolls Royce and Associates Ltd.

2. Fault Analysis and Fault Coverage Code Generation

Fault analysis is the process used to determine the fault detection coverage of a particular electronic circuit design. The fault analysis process for a design involves the optimization of the input stimulus to fully exercise all components to increase the testability, while logic simulation involves the optimization of the functionality of the design. These tasks are very different processes, and both tasks are necessary within the design development process. The fault analysis process fits within the product development cycle after the initial functional verification of the design and before the physical hardware testing of the product.

A statistical fault simulator (i.e. QuickGrade II [10]) is a tool that assists the development of an effective set of *input test vectors* that are passed later to a deterministic fault simulator (i.e. QuickFault II [10]). The statistical fault simulator analyzes the statistics gathered during a logic simulation of the circuit and calculates the probability of detection for each fault in the fault population. The simulator uses the fault detection probabilities in calculating the overall estimated fault coverage for the design. The deterministic fault simulation provides confidence in the comprehensiveness of the test stimulus. The extent to which a set of input vectors can detect real manufacturing faults is called *fault coverage*. If a set of test vectors provides 90 percent fault coverage, the set can detect 90 percent of the faults in that fault population.

The amount of fault coverage within a design depends on the following two factors: (1) comprehensiveness of the test code, and (2) inherent testability of the logic design. In this paper we concentrate on the first factor and formulate the problem of finding an effective set of input test vectors as a *search problem*.

3. The Inductive Genetic Algorithm

In general the inductive approach generates a solution step by step, beginning from the so called *base* of the induction and at each step following an *induction rule* to update (i.e. induce) the solution. In mathematics induction is a rigorous proof technique while in the context of adaptive search it is used to approximately induce a solution to a particular problem. Previous research [11][12] well justifies the potential power of the inductive approach in the context of search.

Applying the inductive approach to the fault coverage problem requires a slight reformuation of the problem. The original problem is:

> Given a number *N* (the maximum number of fault coverage test vectors) find a sequence of *N* test vectors that maximizes the fault coverage.

It can be easily reformulated as:

> For each *k* = 1 to *N* find a sequence of test vectors that maximizes the fault coverage.

While being the same problem the latter formulation also gives meaning to intermediate solutions. Suppose for example that for some k, $1<k<N$, we know a sequence of test vectors which gives *satisfactory* fault coverage. The term satisfactory means that with k test vectors we couldn't expect to cover much more faults than the already discovered by the sequence. Now if for k test vectors we have already achieved a satisfactory level of fault coverage the task is to find a satisfactory fault coverage level for $k+1$ test vectors. The main power in the inductive approach is the assumption that the satisfactory fault coverage level for $k+1$ test vectors could be derived from the satisfactory level of the k test vectors. If this assumption is true then it produces an efficient search engine with computational complexity determined only by the computational complexity of the inductive step.

The overall structure of the inductive search looks as follows:

```
For k = 1 to N do {
      Find the maximal fault coverage achieved by k vectors
      in the context of the maximal fault coverage achieved
      by k-1 vectors
}
```

The Inductive Genetic Algorithm (IGA) combines the evolutionary search engine with an inductive fitness function. The overall structure of the IGA looks as follows:

1. Initialize a partial solution for *N* = 1 (i.e. a sequence of one test vector only)

   100101011011

2. For *k* = 2 to *N* do

3. Initialize a population of test vectors

   [redacted] ... [redacted]

4. Append each test vector to the partial solution, evaluate it and assign fitness

100101011011 ... 100101011011

5. Reproduce according to the fitness obtained in 4. Typical operators include versions of mutation and crossover. Update the population.

6. If not end of generation, goto 4

7. Update the solution, increment k, goto 2

Steps 3, 4, 5, and 6 constitute a genetic algorithm. Steps 1, 2 and 7 implement the inductive approach. The overall algorithm could also be viewed as a genetic algorithm with dynamic fitness function, i.e. the fitness function changes at each generation. The mutation operator randomly selects bits and flips them:

mutation:

```
100101011011    parent chromosome
110111011001    offspring
```

The crossover operator randomly selects bits from two parents and swaps them:

crossover:

```
parent 1     100101011011     011010011001     parent 2
offspring 1  110111011001     000100011011     offspring 2
```

4. Experiments

In order to develop and tune our search strategy we have designed an efficient simple model of the fault population of a *virtual circuit*. As the search engine should be generic enough to run on a variety of circuits such a circuit abstraction comes well justified. In our model each fault population consists of a number of faults with associated fault identifying patterns. A simple fault population with five faults may look like:

Fault No.	Identifying Pattern
1.	*01*********
2.	*01*0*******
3.	101*******10
4.	0*0*11011***
5.	**1*1101****

where "*" is a *don't care* symbol. The task of the fault coverage problem is to find a "generalist" population of test vectors covering as much faults as possible. For example, the following two test vectors cover 80 percent of the fault population:

No.	Test Vector	Fault Covered
1.	101101001110	No. 1, 2, 3
2.	000111011001	No. 4

The associated fitness function with our model is $f = \frac{n}{N} \cdot 100\%$ where n is the number of covered faults and N is the number of all faults in the fault population

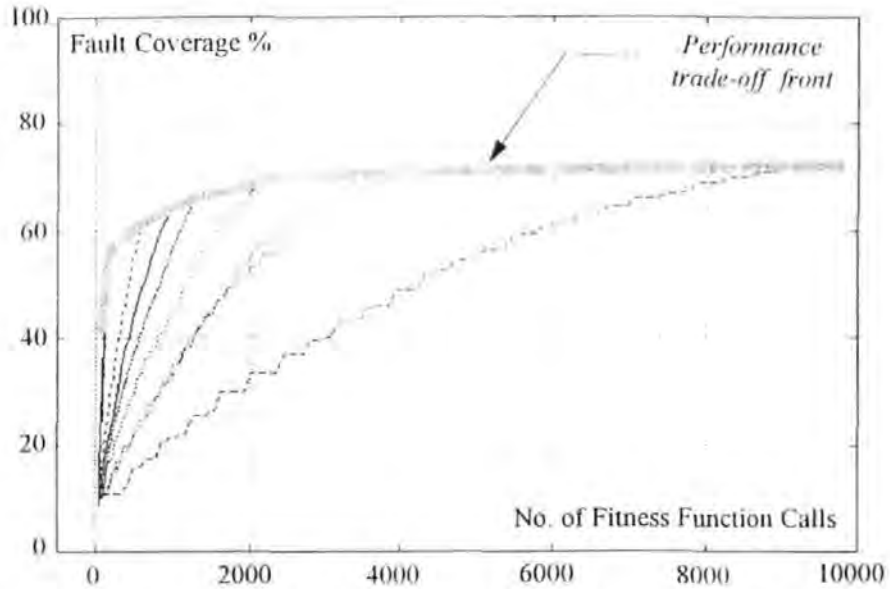


Fig 1: Runs of the Inductive Genetic Algorithm for eight different control parameter settings (population size, mutation rate, crossover rate, number of generations per inductive step). The family of all possible control parameter settings define the *performance trade-off front*, which is a measure of the trade-off between computational complexity and quality of results. The number of input test vectors is twenty four.

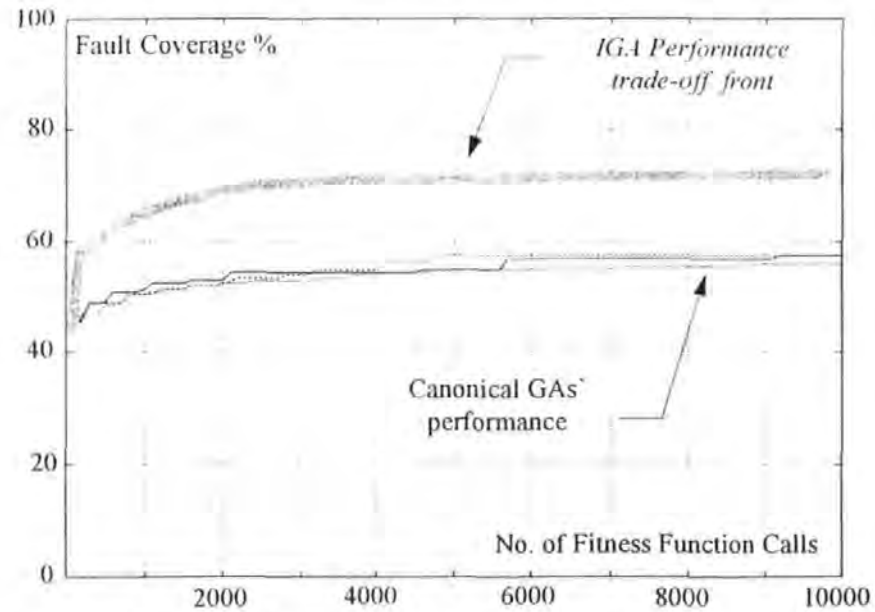


Fig. 2: Comparison between the Inductive Genetic Algorithm (IGA) and the canonical genetic algorithm.

Fig. 1 shows the performance of the Inductive Genetic Algorithm (IGA) on a fault coverage problem consisting of 200 possible faults. The number of input test vectors is twenty four. The search effort at each inductive step controls the trade-off between the computational complexity and the expected quality of results. The family of all possible trade-off points define the

performance trade-off front. It is a measure of the expected gain of the quality of results as a function of the computational expense.

Fig. 2 compares the IGA approach with a cannonical genetic algorithm [1] applied to the whole sequence of test vectors (i.e. when each chromosome has fixed length of 24 vectors 12 bits each and the fitness function is the fault coverage found by the whole set of 24 test vectors). The same test problem as in the previous experiment is used. From fig. 5 it can be clearly seen that the IGA considerably outperforms the classical genetic algorithm. One possible explanation is that the IGA considerably reduces the search space by dividing it into disjoint inductive search subspaces, whereas the classical GA works on the huge original search space (approx. $2^{24 \cdot 12}$).

5. Conclusions

In this paper we have developed a novel evolutionary search approach and applied it to a real world problem. We also discussed circuit abstraction issues. It turns out that making a model of the fault coverage problem provides an efficient and less computationally expensive development framework within which search engines can be developed and tested. This also facilitates the analysis of the search performance. We believe that our approach will result in significant savings of manpower during the test code updating process.

Further work may involve exploration of the salient features of the fault coverage problem. For example, the evolution of "generalist" co-operative population of input test vectors may be tackled by immune networks or clustering genetic algorithms.

References

[1] D. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning", Addison-Wesley, 1989

[2] I. Rechenberg, "Cybernetic Solution Path of an Experimental Problem", Royal Aircraft Establishment Library Translation, Hants, UK, 1965

[3] H. Bersini, and F. Varela, "The Immune Recruitment Mechanism: A Selective Evolutionary Strategy", *TR IR/IRIDIA/91-5*, 1991

[4] D. Dasgupta, and S. Forrest, "Novelty Detection in Time Series Data using Ideas from Immunology", *NIPS-95 Conference*, 22 May, 1995

[5] M. Dorigo, Optimization, Learning, and Natural Algorithms, *PhD Thesis*, Politecnico di Milano, ITALY, 1992

[6] G. Bilchev, and I. Parmee, "Constrained Optimization with and Ant Colony Search Model", *Procs. of 2nd International Conference on Adaptive Computing in Engineering Design and Control*, 28-28 March, University of Plymouth, UK, 1996

[7] D. Wolpert, and W. Macready, "No Free Lunch Theorems for Search", Santa Fe Institute Technical Report SFI-TR-95-02-010, 1995

[8] N. Radcliffe, and P. Surry, "Fundamental Limitations on Search Algorithms: Evolutionary Computing in Perspective", in *Lecture Notes in Computer Science 1000*, Springer-Verlag, 1995

[9] I. Parmee and M. Denham, "The Integration of Adaptive Search Techniques with Current Engineering Design Practice", *Procs. of 1st International Conference on Adaptive Computing in Engineering Design and Control*, University of Plymouth, UK, 1994

[10] Mentor Graphics Corporation, 8005 S.W. Boeckman Road, Wilsonville, Oregon 97070

[11] G. Bilchev, and I. Parmee, "Learning the Next Dimension", *Artificial Intelligence and Simulation of Behaviour'96*, Brighton, UK, April 1996

[12] G. Bilchev, and I. Parmee, "Inductive Search", *First International Contest on Evolutionary Optimization to be held during the 1996 IEEE Conference on Evolutionary Computation*, May 20-22, Nagoya, Japan, 1996

Inductive Search

George Bilchev
Engineering Design Centre
University of Plymouth
Plymouth, PL4 8AA, UK
gbilchev@plymouth.ac.uk

Ian Parmee
Engineering Design Centre
University of Plymouth
Plymouth, PL4 8AA, UK
iparmee@plymouth.ac.uk

Abstract—**In this paper we propose a novel global optimization technique inspired by the process of natural evolution. The employed analogy is that of searching from the simple to the complex, i.e., evolving solutions to problems with gradually increased complexity. The algorithm is applied to the test functions of the First International Contest on Evolutionary Optimization to be held during the 1996 IEEE International Conference on Evolutionary Computation, Nagoya, Japan.**

I. Introduction

Recent growth of interest in search algorithms and search theory [1][2][3] has provoked much debate and controversy. Although one of the problems is that of comparing search algorithms, it is believed that a series of international contests on optimization will shed some light and hopefully answer many of the controversial issues.

For the First International Contest on Evolutionary Optimization we propose a novel global optimization technique inspired by the process of natural evolution. The employed analogy is that of searching from the simple to the complex, i.e., evolving solutions to problems with gradually increased complexity.

Increasing complexity is a salient feature of evolution which we believe is crucial. To justify this idea recall the optimal universal search machine proposed by Levin [4]. It essentially generates and evaluates all solution candidates (strings in a particular representation) in order of their Levin's complexity, until a solution is found. Simpler candidate solutions are generated first and then the search proceeds with increasing the complexity of the newly generated candidate solutions. In evolution, simpler organisms were created first and then the process continues with evolving more and more complex organisms. Utilising the same idea in a computer search context means to evolve a set of solutions to search problems with gradually increasing complexity. One natural way to increase the complexity of a search problem is to increase its dimensions.

Our algorithm is based on the assumption that an approximation of the desired solution can be effectively *constructed* from a limited sample of the search space. The idea is generally borrowed from genetic algorithms and the corresponding *building block hypothesis* [5], but is utilised in a more direct way. We also view the global optimization problem as an existence of short (inductive) rules that can effectively build the solution from a limited sample. If for a particular problem instance such rules do not exist, the global optimization task is not tractable. On the other hand, if such rules exist, the current state of the art is how to find them. In general this problem is not computable. In constrained discrete search spaces (as in our computer models) the problem scales exponentially with the number of dimensions. Historically this phenomenon, known as the "curse of dimensionality", led to the abandonment of direct search methods in favour of ones using some *a priori* knowledge about the cost function f.

A novel feature in our search algorithm is that we do not constrain the sampling procedure to work only in the original search space, but rather divide the problem into a sequence of subproblems and allow sampling in each of the newly defined subspaces. In order to do that our approach requires a *computational model* of the cost function, which is usually available.

The paper is organized as follows: Section 2 describes the proposed evolutionary search algorithm and gives implementation details. Section 3 presents the rationales behind the design of our algorithm. Section 4 shows the experimental results and performance indexes. Section 5 gives conclusions.

II. Proposed Algorithm

The overall structure of our approach is given in fig. 1. For each of the input variables we ask an *oracle* what is the value for which a global optima is achieved. We begin the query from the first variable. Since at this stage the other variables are not yet defined, we consider that they do not exist. For the proposed test functions this is easy to achieve even still treating them as black boxes, because the test functions are defined in terms of two parameters: the number of dimensions and a vector of the input variables:

$$f(1, x_1), f(2, x_1, x_2), \ldots$$

```
main() {
    for (int i=1; i<=NDIM; I++)
        x[i-1]=oracle(i);
}
```

Fig. 1. The overall structure of the proposed approach.

```
void oracle(int dim) {

   sortseq<float,Interval> Population;
   Population.insert(HIGH-LOW, I);

//   Global learning

   while (STOP CONDITION NOT FULFILLED) {

      I = Population.max();
      bren = brent(I.low, (I.low+I.high)/2,I.high,func,TOL,&xmin);
      if (bren < FMIN) { FMIN = bren; XMIN = xmin; }

      if (bx < xmin) {
        I.low = ax; I.high = bx; Population.insert(I.high-I.low, I);
        I.low = bx; I.high = xmin; Population.insert(I.high-I.low, I);
        I.low = xmin; I.high = cx; Population.insert(I.high-I.low, I);
      } else {
        I.low = ax; I.high = xmin; Population.insert(I.high-I.low, I);
        I.low = xmin; I.high = bx; Population.insert(I.high-I.low, I);
        I.low = bx; I.high = cx; Population.insert(I.high-I.low, I);
      }

  }


//   Local learning

   x[dim-1] = XMIN;

   if ( (LEARNING) && (dim > 1) )
      local_learn(func_learning, x, dim);

}
```

Fig. 2. One possible implementation of the oracle. LOW is the lower bound of the search space and HIGH is the upper bound. TOL is a tolerance for the Brent's routine, which is set to 1E-4.

At a later stage, i.e., when solving $f(i,\ldots,x_i)$, the oracle can "update" a previous answer by changing the values of x_1 to x_{i-1}. This is necessary, because when the oracle solves $f(i-1,\ldots,x_{i-1})$ it has no knowledge of how this function will be upgraded to $f(i,\ldots,x_{i-1},x_i)$.

If such an oracle was easy to find and implement, then the global optimization problem would have been trivial. However, it is usually hard to decide how to define the oracle procedure. In this paper, we have implemented a simple deterministic version of the oracle which proved to perform very well on the proposed test functions. The basic algorithm is given in fig. 2.

It uses two distinct steps. First it searches for good solutions of the current dimension in the context of the best solution(s) of the previous dimensions (*global learning*).

Then it locally searches in all dimensions starting from the already discovered "promising" areas (*local learning*).

In the proposed implementation, the global learning is a series of calls to Brent's parabolic interpolation routine:

```
float brent(float ax, float bx,
   float cx, float (*f)(float),
   float tol,float *xmin);
```

where, given a function `f`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` (such that `bx` is between `ax` and `cx`, and `f(bx)` is less than both `f(ax)` and `f(cx)`), this routine isolates the minimum to a fractional precision of about `tol` using Brent's method. The abscissa of the minimum is returned as `xmin`, and the minimum function value is returned as `brent`, the returned function value [6]. In an attempt to minimize the maximal risk to miss a solution, the algorithm always applies the Brent's routine to the largest interval in the population.

The local learning is implemented as dynamic hill-climbing:

```
local_learn(float (*f)(float*x),
    float x,int dim);
```

where, given a function `f`, a starting point `x`, and number of dimensions `dim`, ***local_learn*** finds a local optima [7]. Initial step size of `1.0` and threshold of `1E-4` are used.

Various stopping conditions are possible. We have tested value to reach (i.e., stop the global learning when the algorithm reaches a predetermined value) and a maximal number of calls to Brent's routine.

III. RATIONALES

It is a well established approach to employ analogies from nature for problem solving. Examples include simulated annealing, genetic algorithms, immune networks, etc. In the area of evolutionary computation the most widely used algorithms usually utilise the survival of the fittest principle. It remains surprising how little attention has been given to the fact that evolution goes from the simple to the complex, i.e., simpler organisms are created first and then they evolve and become more and more complex. This interpretation is quite close to the description of the optimal universal search machine originally proposed by Levin [4]. Although it has been proven that the optimal universal search algorithm is not computable, it may be that evolution is one of its most fascinating implementations.

To utilise the same idea in a computer search context we need to define a family of complexity classes for the particular problem at hand. One way to do it is by increasing the dimensionality of the problem.

It is a well established phenomenon that with the increase of the number of dimensions the search problem scales exponentially ("the curse of dimensionality"). This historically led to the abandonment of direct search methods in favour of those utilising some *a priori* knowledge about the cost function. Recently the No Free Lunch theorem [1] argued that incorporating *a priori* knowledge into the search process is the only legitimate way to overcome the fundamental limitations on search algorithms [2].

However, there is a strong counter argument against the NFL theorem line of reasoning. It is that most of the cost functions do not have short descriptions, i.e., the only way to represent them is by a table of (x,y) pairs, which given the size of the search space is practically impossible. Only a small fraction of all the possible cost functions have short descriptions and therefore, are able to be implemented as a computer model. It is true that these functions have various different properties, i.e., multi-modality, smoothness, etc. However, they share the short description (i.e., low algorithmic complexity) property. It also seems quite probable that all these functions share another property, namely short descriptions of the global optima. The short description of the global optima could be used as a kernel of a tractable search algorithm. It is clear that the short description assumption is not too binding as far as practical computer models are concerned, so any search algorithm which makes such an *a priori* assumption remains of great practical importance.

The assumption made by our algorithm is that the global minimum of $f(N, x_1, \ldots, x_N)$ could be found by finding the global minimum of $f(N, \hat{x}_1^*, \ldots, \hat{x}_{N-1}^*, x_N)$ in the context of the global minimum of $f(N-1, \hat{x}_1^*, \ldots, \hat{x}_{N-1}^*)$ and then learning the N variables that match the global optimum of $f(N, x_1, \ldots, x_N)$. In other words the assumption is that:

$$\Omega\left(\hat{x}_1^*, \ldots, \hat{x}_{N-1}^*, \tilde{x}_N^*\right) = \left(x_1^*, \ldots, x_N^*\right)$$

where Ω is a polynomial time learning operator, $\left(\hat{x}_1^*, \ldots, \hat{x}_{N-1}^*\right)$ is the global optimum of $f(N-1, x_1, \ldots, x_{N-1})$, $\tilde{x}_N^*$ is the global optimum of $f(N, \hat{x}_1^*, \ldots, \hat{x}_{N-1}^*, x_N)$, and $\left(x_1^*, \ldots, x_N^*\right)$ is the global optimum of $f(N, x_1, \ldots, x_N)$.

The class of functions for which the above assumption is true do not suffer from the "curse of dimensionality". Inducing a new dimension is achieved by solving only one and the same (type of) one dimensional global optimization problem followed by a polynomial time learning. This assumption seems to be true for all of test beds proposed by the organizing committee of the First International Contest

on Evolutionary Optimization. It has been also shown that for some of the most widely used test functions in the evolutionary computing community the scaling is even linear [8].

IV. Experiments

In order to better explain our algorithm we first run through its basic steps using the following test case (Langerman's test function):

$$f(\vec{x}) = -\sum_{i=1}^{m} c_i \left(e^{-\frac{1}{\pi}\|\vec{x}-A(i)\|^2} \cdot \cos\left(\pi \cdot \|\vec{x}-A(i)\|^2\right)\right), \quad m = 5$$

Initially, ***brent*** is used to find the set of local minima of the 1-D version of the test function (fig. 3):

$x_1^1 = 2.197, \quad f(x_1^1) = -0.908$
$x_1^2 = 6.628, \quad f(x_1^2) = -1.276$
$x_1^3 = 8.184, \quad f(x_1^3) = -2.709$
$x_1^4 = 9.450, \quad f(x_1^4) = -2.579$

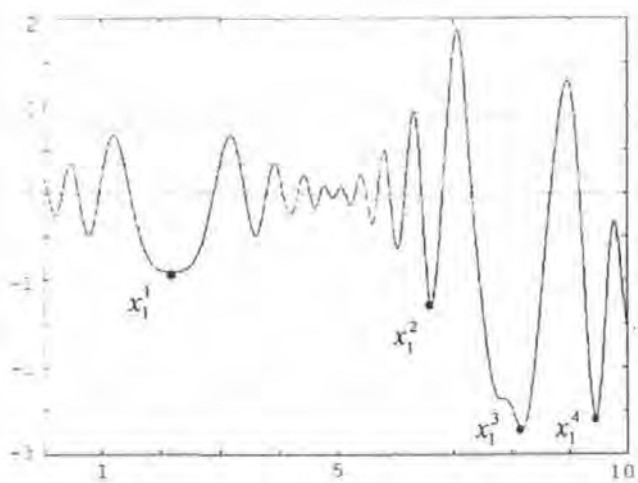


Fig. 3. Langermans function for N=1.

According to our assumption, we get $x_1^* = x_1^3$ and use ***brent*** to find the set of local minima of $f_2(x_1^*, x_2)$ (fig. 4):

$x_2^1 = 0.489, \quad f(x_1^*, x_2^1) = -0.396$
$x_2^2 = 6.746, \quad f(x_1^*, x_2^2) = -0.432$
$x_2^3 = 7.237, \quad f(x_1^*, x_2^3) = -0.425$
$x_2^4 = 8.995, \quad f(x_1^*, x_2^4) = -2.389$

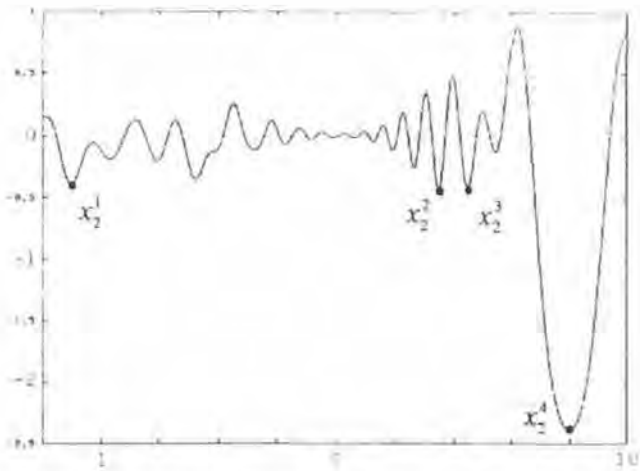


Fig. 4. One dimensional version of Langerman's function for N=2, where $x_1^* = x_1^3$ (fig. 4)

Next, ***local_learn*** is used starting from $f(x_1^*, x_2^*)$, where $x_2^* = x_2^4$. It updates the values of x_1^* and x_2^* to 8.047 and 8.985 respectively. An overview of $f_2(x_1, x_2)$ is given in fig. 5. The new values are used in the next iteration, where ***brent*** seeks the set of local optima of $f(x_1^*, x_2^*, x_3)$ The process is further iterated.

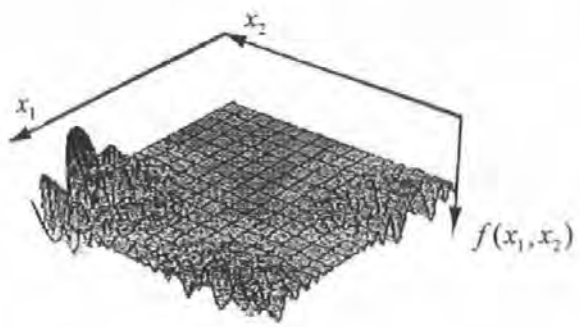


Fig. 5. Langerman's function for N=2.

The test problems used in the Global Optimization community can be characterized as essentially unconstrained problems with cheap to evaluate objective functions, having analytical derivatives and a small number of minima with $\max_i p_i = p_1$, $(p_1 \geq 0.2)$ in a low dimensional decision space (where p_1 is the probability to sample the basin of attraction of the global optima). The importance of a given approach to optimization depends above all on the practical

problems which may be efficiently solved by the proposed algorithms. Therefore, it is advisable to use real world problems in validating all the stages of development of algorithms, starting with the justification of main theoretical assumptions. However, it is difficult to use practical problems in investigating and testing algorithms because the practical objective functions are usually expensive to evaluate and quite often practical problems cannot be freely distributed. Therefore, normally some artificial test functions are used to test the algorithms.

The results on the proposed test functions for the first international contest on evolutionary optimization are shown in table 1. The third column tells whether a local learning is employed, and the maximal number of calls to Brent's routine at each dimension is presented in column four.

f	dim	L	ΔN	value	No of calls
Sp	5	*no*	1	0	20
Sp	10	*no*	1	0	40
Gr	5	*no*	1	0	210
Gr	10	*no*	1	0	420
Sh	5	*yes*	2	-10.404	388
Sh	10	*yes*	2	-10.2079	796
Mi	5	*no*	3	-4.688	386
Mi	10	*no*	10	-9.660	1744
La	5	*yes*	3	-1.5	558
La	10	*yes*	3	-1.5	958

Table 1: Results for the test problems: Sp: Sphere model, Gr: Griewank's function, Sh: Shekel's foxholes, Mi: Michalewicz's function, La: Langerman's function.

When, instead of maximal number of calls to Brent's routine we stop the global learning upon reaching a predetermined value (value to reach), the following results are obtained:

f	dim	value	No of calls
Sp	5	3.88e-15	20
Sp	10	7.10e-15	40
Gr	5	7.99e-6	41
Gr	10	1.31e-6	79
Sh	5	-10.327	74
Sh	10	-10.101	120
Mi	5	-4.688	120
Mi	10	-9.660	501
La	5	-1.499	176
La	10	-1.499	372

Table 2: Results for the test problems: Sp: Sphere model, Gr: Griewank's function, Sh: Shekel's foxholes, Mi: Michalewicz's function, La: Langerman's function.

V. Conclusions

In this paper we present a novel algorithm for global optimization inspired by natural evolution. It is based on the assumption of polynomial time induction of the "next" dimension. During the inductive step we have utilized a local hill climber which is well justified by the nature of the proposed test functions. However, it is easy to show that a local hill climber could not always induce the next dimension. This is especially true for functions which introduce new information in the subsequent dimensions. The new information could potentially "invalidate" the results obtained so far. The invalidation is relative to the employed learning algorithm. If the test function has a short description, so has the law for introducing the new information. An ability to identify this law in polynomial time will again lead to polynomially inducible new dimensions. Further research into this area is required.

References

[1] D. Wolpert, and W. Macready, No Free Lunch Theorems for Search, Santa Fe Institute Technical Report SFI-TR-95-02-010, 1995

[2] N. Radcliffe, and P. Surry, Fundamental Limitations on Search Algorithms: Evolutionary Computing in Perspective, *LNSC 1000*, Springer-Verlag, 1995

[3] H. Kargupta, SEARCH, Polynomial Complexity, And the Fast Messy Genetic Algorithm, PhD Thesis, Univ. of Illinois At Urbana-Champaign, IlliGAL Report No. 95008, 1995

[4] L.A. Levin, Universal sequential search problems, *Problems of Information Transmission*, 9(3):265-266, 1973

[5] Goldberg D. E., Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, Reading, MA, 1989

[6] Press W., Teukolsky S., Vetterling W., and Flannery B., Numerical Recipes in C, Cambridge University Press, 1992

[7] D. Yuret, From Genetic Algorithms to Efficient Optimization, *MSc. thesis*, MIT, May 1994

[8] G. Bilchev, and I.C. Parmee, Learning the "Next" Dimension, *AISB '96*, Brighton, UK