Searching the Solution Space in Constructive Geometric Constraint Solving with Genetic Algorithms

Article *in* Applied Intelligence · March 2005

DOI: 10.1007/s10489-005-5600-1 · Source: DBLP

4 authors, including:

Maria Victoria Luzon
University of Granada
40 PUBLICATIONS **395** CITATIONS
SEE PROFILE

J.F Galvez
University of Vigo
26 PUBLICATIONS **358** CITATIONS
SEE PROFILE

Robert Joan-Arinyo
Universitat Politècnica de Catalunya
76 PUBLICATIONS **998** CITATIONS
SEE PROFILE

Searching the Solution Space in Constructive Geometric Constraint Solving with Genetic Algorithms

M.V. Luzón[1], A. Soto[2], J.F. Gálvez[1], R. Joan-Arinyo[2]

[1]Escuela Superior de Ingeniería Informática.
Universidade de Vigo,
Av. As Lagoas s/n, E-32004 Ourense
{luzon,galvez}@uvigo.es

[2]Escola Técnica Superior d'Enginyeria Industrial de Barcelona.
Universitat Politècnica de Catalunya,
Av. Diagonal 647, 8^a, E-08028 Barcelona
{robert,tonis}@lsi.upc.es

Abstract

Geometric problems defined by constraints have an exponential number of solution instances in the number of geometric elements involved. Generally, the user is only interested in one instance such that besides fulfilling the geometric constraints, exhibits some additional properties. Selecting a solution instance amounts to selecting a given root every time the geometric constraint solver needs to compute the zeros of a multi valuated function. The problem of selecting a given root is known as the *Root Identification Problem*.

In this paper we present a new technique to solve the root identification problem. The technique is based on an automatic search in the space of solutions performed by a genetic algorithm. The user specifies the solution of interest by defining a set of additional constraints on the geometric elements which drive the search of the genetic algorithm. The method is extended with a sequential niche technique to compute multiple solutions. A number of case studies illustrate the performance of the method.

Keywords Genetic algorithms, Constructive geometric constraint solving, Root identification problem, Solution selection.

1 Introduction

Nowadays the design and manufacture of industrial products is an increasingly competitive market. In this setting, the Computer-Aided Design and Manufacture (CAD/CAM) systems must help to improve efficiency and, significantly shortening the design-to-manufacture cycle. To achieve this goal, CAD/CAM systems should provide tools to support for conceptual design and to capture the design intent, that is, what the user has in mind.

Contemporary CAD/CAM systems capture the design intent partially by defining functional relationships between dimensional variables, that is parametric design, and constraint-based geometric design. In constraint-based geometric design, the designer creates a rough sketch of an object composed of simple geometric elements. Then the intended exact shape is specified by annotating the sketch with constraints. A geometric constraint system, known as *solver*, checks whether the set of geometric constraints consistently defines the object and, if so, determines the position of the geometric elements.

Geometric problems defined by constraints have a number of solution instances which is exponential in the number of geometric elements involved. Among these many solution instances, generally, the user is interested in just one that fulfills the geometric constraints and whose shape is *similar* to the design intent. Therefore, intelligent CAD/CAM systems should provide with tools to properly select the instance solution the user expects to get, if one exists.

Conceptually, selecting a solution instance amounts to selecting one among a number of different roots of a nonlinear equation or system of equations. The problem of selecting a given root was named in [1] the *Root Identification Problem*.

Several approaches to solve the root identification problem have been reported in the literature. In an old approach, the user interactively selects geometric elements that should be moved to specific positions. Then the system checks for the feasibility by cheking whether the constraints still hold. Clearly, this is a trial-and-error method.

A simple approach conducts a dialogue between the user and the constraint solver to identify interactively the intended solution by navigating in the space of instance

solutions. Given the potentially exponential number of instance solutions in this space, the approach is of little practical interest.

A common assumption in the CAD/CAM community is that users realize their designs by scketching shapes that are close to what they expect to get. Based on this assumption, the preserving topology strategy selects the intended solution instance by automatically placing the geometric elements in such a way that their relative positions are the same as in the sketch. Drawbacks of this approach are that such a solution instance is not necessarily unique or that it not necessarily exists. For a discussion of these approaches see, for example, references [1, 2, 3, 4] and references therein.

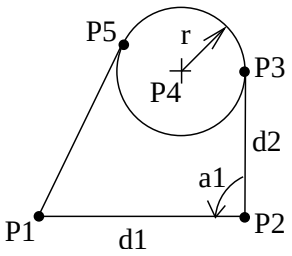
Adding extra constraints to narrow down the number of possible solutions to constraint geometric problems seems to be a simple approach. However, this approach has been carefully avoided in the field because the resulting over-constrained problem is NP hard. Moreover, the set of constraints may be contradictory, [1].

In this paper we present a new technique to automatically solve the root identification problem by over-constraining the geometric constraint problem. Over-constrained situations are avoided by defining the geometric constraints as belonging to two different categories. One category includes the set of constraints needed specifically to solve the geometric constraint problem. The other category defines a set of extra constraints or predicates on the geometric elements which identify the intended solution instance.

Once the solver has generated the space of solucion instances, the extra constraints are used to drive an automatic search of the solution instances space performed by a genetic algorithm, [5, 6]. The search outputs a solution instance that maximizes the number of extra constraints fulfilled.

The approach has been implemented and tested, [4]. Experimental results show that the approach is both effective, that is, whenever the intended solution instance selected exists it is found; and efficient, the number of solutions instances actually checked is always a small fraction of the potentially exponential number of solution instances. The basic technique is then extended with a *niching method* to select and maintain multiple solutions.

The outline of the rest of the paper is as follows. In Section 2 we briefly review the basic concepts of constructive geometric constraint solving. Section 3 is devoted to the



distance(P1, P2) = d1
distance(P2, P3) = d2
angle(segment(P2, P3), segment(P1, P2)) = a1
tangent(segment(P2, P3), circle(P4, r)
on(P3, circle(P4, r))
tangent(segment(P1, P5), circle(P4, r))
on(P5, circle(P4, r))

Figure 1: Geometric problem defined by constraints.

solution instance selector based on the genetic algorithm. The niche extension is presented in Section 4. As a proof of concept, we present in Section 5 some experimental results. Finally, Section 6 offers a summary and open questions for future work.

2 Constructive Geometric Constraint Solving

In two-dimensional constraint-based geometric design, the designer creates a rough sketch of an object made out of simple geometric elements like points, lines, circles and arcs of circle. Then, the intended exact shape is specified by annotating the sketch with constraints like distance between two points, distance from a point to a line, angle between two lines, line-circle tangency and so on. A geometric constraint solver then checks whether the set of geometric constraints coherently defines the object and, if so, determines the position of the geometric elements.

If geometric elements and constraints are like those above, a constraint-based design can be represented by a set of points along with a set of constraints drawn from distance between two points, distance from a point to a line, and angle between two lines, [7]. Figure 1 shows an example sketch of a constraint-based design.

2.1 A Formalization of the Geometric Constraint Problem

It is well known that the relative position of n given points $\{p_1, p_2, \ldots, p_n\}$ in the bidimensional Euclidian space, is determined by $2n - 3$ independent relationships defined between the points, [8, 9]. Based on this fact, the geometric constraint problem in the

Euclidean space can be formalized as follows.

First we assume that a given set of n points, on which a set of $2n-3$ independent constraints has been defined, is split into two nonempty disjoint subsets. One subset, $\vec{p'} = \{p'_1, p'_2, \ldots, p'_k\}$, contains all those given points with fixed position. The other subset, $\vec{p} = \{p_1, p_2, \ldots, p_l\}$, contains all those points with unknown position. Notice that this decomposition is always possible because $2n-3$ independent relationships between n given points define a rigid body with three remaining degrees of freedom, two of them corresponding to a translation and the third one corresponding to a rotation. Hence, the absolute position for at least one given point should be specified.

Following Brüderlin [10], the set of constraints along with logical conjunction, disjunction and negation allow us to express the geometric constraint problem by a first order logic formula $\varphi(\vec{p'}, p_1, \ldots, p_l)$ such that if the set of constraints defines a well constrained problem, [11, 12], the formula

$$\exists p_1 \ldots \exists p_l \; \varphi(\vec{p'}, p_1, \ldots, p_l)$$

holds. By the *axiom of choice*, [10, 13], we can say that whenever the above formula holds, the formula

$$\exists f_1 \ldots \exists f_l \; \varphi(\vec{p'}, f_1(\vec{p'}), \ldots, f_l(\vec{p'}))$$

also holds. Hence, the goal in solving a geometric constraint problem is to prove the truth of the above formula, and to evaluate the functions $f_1, \ldots, f_l$.

As we will be seen in the next Section, the constructive geometric constraint solving approach is such that given the geometric constraint problem $\varphi(\vec{p'}, p_1, \ldots, p_l)$ defined on the set points $\{p_1, p_2, \ldots, p_n\}$, searches a *constructive* first order logic formula, $\Psi(\vec{p'}, p_1, \ldots, p_l)$, such that, if the constraint problem is well constrained, will figure out the relative position of each point. If the predicate $pos(p_i, (x_i, y_i))$ assigns the position (x_i, y_i) to point p_i, an example of the constructive formula would be, [10],

$$\begin{aligned}\Psi(\vec{p'}, p_1, \ldots, p_l) \quad = \quad & pos(p'_1, (x_1, y_1)) \wedge \ldots \wedge pos(p'_k, (x_k, y_k)) \wedge \\ & pos(p_1, (fx_1(\vec{p'}), fy_1(\vec{p'}))) \wedge \\ & \bigwedge_{i=2}^{l} pos(p_i, (fx_i(\vec{p'}, p_1, \ldots, p_{i-1}), fy_i(\vec{p'}, p_1, \ldots, p_{i-1})))\end{aligned}$$

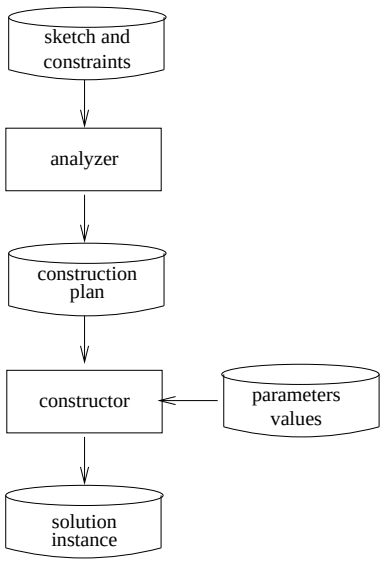


Figure 2: Basic architecture of constructive geometric constraint solvers.

2.2 Solving the Geometric Constraint Problem

Many techniques have been reported in the literature that provide powerful and efficient methods for solving systems of geometric constraints. For example, see [14] and references therein for an extensive analysis of the work done on constraint solving. Among all the geometric constraint solving techniques, our interest focuses on those known as *constructive*.

Constructive solvers have two major components: the *analyzer* and the *constructor*. The analyzer symbolically determines whether a geometric problem defined by constraints is solvable. If the problem is solvable, the output of the analyzer is a sequence of construction steps each of them corresponding to a pair of functions (fx_i, fy_i) in the above first order logic formula which places each geometric element in such a way that all constraints are satisfied. This sequence is known as the *construction plan*. After assigning specific values to the parameters, the constructor interprets the construction plan and builds an object instance, provided that no numerical incompatibilities arise. Figure 2 illustrates the main components in a constructive geometric constraint solver. See [15] for a declarative characterization.

$$
\begin{aligned}
P_1 &= point(0,0) \\
P_2 &= point(d_1,0) \\
\alpha_1 &= direction(P_1,P_2) \\
\alpha_2 &= adif(\alpha_1,a_1) \\
P_3 &= rc(line(P_2,\alpha_2),circle(P_2,d_2),s_1) \\
\alpha_3 &= direction(P_2,P_3) \\
\alpha_4 &= asum(\alpha_3,\pi/2) \\
Q_1 &= rc(line(P_2,\alpha_4),circle(P_2,r),s_2) \\
P_4 &= rc(line(Q_1,\alpha_3),circle(P_3,r),s_3) \\
Q_2 &= midpoint(P_1,P_4) \\
r_1 &= distance(P_1,Q_2) \\
P_5 &= cc(circle(P_4,r),circle(Q_2,r_1),s_4)
\end{aligned}
$$

Figure 3: Construction plan for the object in Figure 1.

The specific construction plan generated by an analyzer depends on the underlying constructive technique and on how it is implemented. For example, the ruler-and-compass constructive approach is a well-known technique where each constructive step in the plan corresponds to a basic operation solvable with a ruler, a compass and a protractor. In practice, this simple approach solves most useful geometric problems. Figure 3 shows a construction plan for the object of Figure 1, generated by the ruler-and-compass geometric constraint solver reported in [16].

Function names in the plan are self explanatory. For example function $adif$ denotes subtracting the second angle from the first one and $asum$ denotes the addition of two angles while rc and cc stand for the intersection of a straight line and a circle, and the intersection of two circles, respectively.

Functions (fx_i, fy_i) in the specific constructive first order logic formula for this example, $\Psi(P_1, P_2, P_3, P_4, P_5)$, are obtained from the construction plan in Figure 3 by

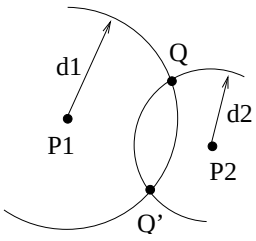


Figure 4: Possible placements of a point.

replacing symbols on the left of the equal sign different from P_i with their definitions. For example, in α_2 replace α_1 with $direction(P_1, P_2)$ and plug in P_3 the resulting expression for α_2.

In general, a well constrained geometric constraint problem, [17, 12, 8], has an exponential number of solutions. For example, consider a geometric constraint problem that properly places n points with respect to each other. Assume that the points can be placed serially, each time determining the next point by two distances from two already placed points. In general, each point can be placed in two different locations corresponding to the intersection points of two circles, (see Figure 4). For n points, therefore, we could have up to 2^{n-2} solutions.

Different possible locations of geometric elements corresponding to different roots of systems of nonlinear algebraic equations can be distinguished by enumerating the roots with an integer index. For a more formal definition see [3, 18].

In what follows, we assume that the set of geometric constraints coherently defines the object under design, that is, the object is generically well constrained and that a ruler-and-compass constructive geometric constraint solver like that reported in [16] is available.

In this solver, intersection operations where circles are involved, rc and cc, may lead to up to two different intersection points, depending on whether the second degree equation to be solved has no solution, one or two different solutions in the real domain. With each feasible rc and cc operation, the constructor in the solver associates an integer parameter $s_k \in \{-1, 1\}$, that characterizes each intersection point by the sign of the square root in the corresponding quadratic equation. In the construction plan given

in Figure 3, the set of indices includes s_k, $1 \leq k \leq 4$. Notice that if a set of values are assigned to the constraint parameters, d_1, d_2 and a_1, the construction plan defines the space of solution instances as a function of s_k. For details on how to compute s_k, the reader is referred to [19] and [18].

2.3 The Root Identification as a Constraint Optimization Problem

In the technique presented in this work, the root identification problem is solved by over constraining the geometric constraint problem. The technique has two different steps. In the first step, the user defines the set of constraints. To avoid over-constrained situations, geometric constraints are defined as belonging to two different categories. One category includes the set of constraints needed to specifically solve the geometric constraint problem at hand. The other category includes a set of extra constraints or predicates on the geometric elements with which the user identifies the intended solution instance.

In the second step, the solver generates the space of solution instances, as a construction plan. Then, a search of the solution instances space is performed seeking for a solution instance which maximizes the number of extra constraints fulfilled. In this context, solving the root identification problem amounts to solving a general constraint-satisfaction problem expressed as a constraint optimization problem.

Genetic algorithms have proven to be an effective technology for solving general constraint-satisfaction problems, [30, 6]. With the aim of applying genetic algorithms, we show how our root identification problem can be formalized as a constraint optimization problem. Recall from Section 2.2 that we consider ruler-and-compass constructive geometric constraint solving where geometric operations correspond to quadratic equations, thus each constructive step has at most two different roots.

Let s_j denote the integer parameter associated by the solver with the j-th intersection operation, either rc or cc, occurring in the construction plan. Since we are interested only in solution instances that are actually feasible, that is, solution instances where no numerical incompatibilities arise in the constructor, we only need to consider integer parameter s_j taking value in the set of signs $D_j = \{-1, 1\}$ that characterizes

each intersection point.

Assume that n is the total number of rc plus cc intersection operations in the construction. We define the *index* associated with the construction plan as the ordered set $I = \{s_1, \ldots, s_j, \ldots, s_n\}$ with $s_j \in D_j, 1 \leq j \leq n$. Therefore the cartesian product of sets $\mathcal{I} = D_1 \times \ldots \times D_n$ defines the space where the solution instances to the geometric constraint problem belong to.

A construction plan which is a solution to a geometric constraint problem can be seen as a function of the index I. Let $\Psi(I)$ denote the construction plan expressed as the corresponding constructive first order logic formula defined in Section 2.1 and illustrated in Section 2.2. Clearly, the set of indices $\{I \in \mathcal{I} \mid \Psi(I) = true\}$ is the space of feasible indices, that is the set of indices each selecting one solution to the geometric constraint problem. This set of indices is the *allowable search space*, [30].

Let Φ denote the first order logic formula defined by the conjunction of the extra constraints given to specify the intended solution instance. Let f be a (possibly real-valued) function defined on $\Psi(I) \wedge \Phi$ which has to be optimized. Then, according to Eiben and Ruttkay, [30], the triple $< \mathcal{I}, f, \Psi(I) >$ defines a *constraint optimization problem* where finding a solution means finding an index I in the allowable search space with an optimal f value.

In simple terms, the problem we are studying can be stated as follows:

> Let G be a geometric problem defined by a set of constraints C. Let $P(\mathcal{I})$ be a construction plan generated by a constructive solver that defines the space of instance solutions to G as a function of the index $\mathcal{I}$. Let C' be the set of extra constraints that characterizes the intended solution instance. Find an index $I \in \mathcal{I}$ such that the number of constraints in C' fulfilled by the solution instance $P(I)$ is maximum.

3 The Genetic Algorithm for the Root Identification

Once the root identification problem by over-constraining the constraint problem has been transformed into a constraint optimization problem, the usual machinery in ge-

netic algorithms can be applied to find a solution. We first recall some basic concepts from genetic algorithms. Then we present how we apply them to solve our problem.

3.1 Basic Background on Genetic Algorithms

Evolutionary Computation uses computational models of evolutionary processes as key elements in the design and implementation of computer-based problem solving systems. There are a variety of evolutionary computational models that have been proposed and studied which are referred to as Evolutionary Algorithms. There have been four well-defined evolutionary algorithms which have served as the basis for much of the activity in the field: Genetic Algorithms (GA), [5, 21], Genetic Programming, Evolution Strategies, [22, 23] and Evolutionary Programming, [24, 25].

An evolutionary algorithm maintains a population of trial solutions, imposes random changes to these solutions, and incorporates selection to determine which ones are going to be maintained in future generations and which will be removed from the pool of the trials. GAs emphasize models of genetic operators as observed in nature, such as crossover (recombination) and point mutation, and apply these to abstracted chromosomes. Evolution strategies and evolutionary programming emphasize mutational transformations that maintain the behavioral linkage between each parent and its offspring.

GAs, [5], are theoretically and empirically proven algorithms that provide a robust search in complex spaces, thereby offering a valid approach to problems requiring efficient and effective searches.

Any GA starts with a population of randomly generated solutions, called chromosomes, and advances toward better solutions by applying genetic operators, modeled according to genetic processes occurring in nature. In these algorithms we maintain a population of solutions for a given problem; this population undergoes evolution in the form of natural selection. In each generation, relatively good solutions reproduce to give offspring that replace the relatively bad solutions which die. An evaluation or fitness function plays the role of the environment to distinguish between good and bad solutions. The process of going from the current population to the next population

constitutes one generation in the execution of a GA.

The use of genetic algorithms has been instrumental in achieving good solutions to discrete problems that have not been satisfactorily addressed by other methods, [5]. Recent surveys can be found in [5] and [26].

Genetic Programming, [27, 28], is a domain-independent approach to automatic programming in which computer programs are evolved to solve, or approximately solve, problems.

3.2 The Genetic Algorithm

In what follows we will use the terms *solution instance* and *intended solution instance*. A *solution instance* to the geometric constraint problem is an individual I in the allowable search space where the Boolean formula $\Psi(I)$ holds, that is, an individual which actually defines a solution to the geometric constraint problem.

An *intended solution instance* to the geometric constraint problem is a solution instance for which all the extra constraints hold, that is, the Boolean formula Φ holds.

Evolutionary algorithms which model natural evolution processes were already proposed for optimization in the 1960s. The goal was to design powerful optimization methods, both in discrete and continuous domains, based on searching methods on a population of coded problem solutions, [20].

The genetic algorithm we have implemented is given in Algorithm 1. P is the population of individuals at the current generation. It consists on a fixed, given number of individuals in $\mathcal{I}$. The main components of the genetic learning process are described as follows.

3.2.1 Initial Gene Pool

As stated in Section 2, the analyzer generates a construction plan that symbolically determines whether a geometric problem defined by constraints is solvable. But the construction plan does not provide any specific information about any index of any solution instance. Therefore, the initial population is randomly generated.

Procedure GeneticAlgorithm

INPUT

F : Functions in the construction plan.

C : Values actually assigned to the constraints.

R : Set of extra constraints.

ng : Number of generations.

OUTPUT

I : Index selected.

InitializeAtRandom (P)

Evaluate(P, F, C, R)

I = SelectCurrentBestFitting (P)

while not TerminationCondition (ng, I, R)

do

Selection (P)

Crossover (P)

Mutation (P)

ApplyElitism (P, I)

Evaluate(P, F, C, R)

I = SelectCurrentBestFitting (P)

$ng = ng + 1$

endwhile

return I

EndProcedure

Algorithm 1: Genetic algorithm.

3.2.2 Evaluating the Chromosome

As stated in Section 3.1, in a given constraint optimization problem, $< \mathcal{I}, f, \Psi(I) >$, the function f, defined over $\Psi(I) \wedge \Phi$, is the goal to be optimized by the genetic algorithm. It is called the *fitness function*.

In general, constraints are handled by constructing f as a summation of penalty terms which penalize the fitness of individuals in the population, according to the degree of violation of the constraints in $\Psi(I) \wedge \Phi$.

Designing an appropiate penalty function that enables the genetic algorithm to converge to a feasible suboptimal or even optimal solution is crucial, [31], and entails addressing two issues. One is to define the relative penalty with which each constraint contributes to the overall fitness function. The other is decoupling the fitness function from the specific genetic technique applied. In general, addressing these issues in an effective way requires substantial knowledge on the problem at hand, [32].

To alleviate this drawback, genetic algorithms with varying fitness functions have been developed. See, for example, [32] and [33] and the references therein. Varying fitness functions make use of the natural adaptive behavior of evolutionary algorithms based on the fact that they dynamically adjust certain parameters according to the evolution of past experiences. As a result, locating the region where the global optimum is located in the search space is favoured, [33].

We carried out our experiments using a very simple non varying fitness function. The fitness of each chromosome I in the population was measured just by counting the number of additional geometric constraints fulfilled by the individual:

$$f(I) = \begin{cases} \sum_{i=1}^{|R|} \delta(R_i(I)) & \text{if } I \text{ is a solution instance} \\ \\ MIN & \text{otherwise} \end{cases}$$

where $\delta(R_i(I)) = 1$ if the solution instance associated with chromosome I fulfills the extra constraint R_i in Φ, and $\delta(R_i(I)) = 0$ otherwise. That is, to evaluate an chromosome fitness involves counting how many extra constraints its associated solution instance fulfills. MIN is the minimum fitness value in the previous generation. The output of the genetic algorithm is an individual that maximizes the fitness function.

As we will illustrate in Section 5, the search space is sparse because the ratio between the number of indices which define intended solutions and the total number of potential solutions is small. However, contrarily to what is reported in [6] and [31], the non varying fitness function did not make the genetic algorithm to fail.

3.2.3 Genetic Search Operators

Search strategies in genetic algorithms are built using a set of constructive genetic search operators. Each operator provides a different scope to the search process, [34]. The set of genetic operators we have considered includes: Selection with elitism, crossover (recombination) and mutation.

Selection

Selection is the process of choosing individuals for reproduction. The selection technique chosen has an effect on the genetic algorithm convergence. If too many individuals with vastly superior fitness are selected, the algorithm can converge prematurely. If too few individuals with vastly superior fitness are selected, algorithm convergence toward optimal solutions would be too slow. Two different selection strategies were applied: Proportional selection and linear ranking selection.

Proportional selection, [35], assigns a reproductive probability to each individual that is proportional to the individual's relative fitness. If $f(I_i)$ is the fitness of chromosome I_i, and N is the number of individuals in the current population, the probability of selecting I_i is given by

$$p_s(I_i) = \frac{f(I_i)}{\sum_{j=1}^{j=N} f(I_j)}$$

Then individuals are selected by the procedure commonly called the *roulette wheel* sampling algorithm, [35]. To preserve the best solution instance in each generation we applied the simplest elitism technique consisting on keeping just the best individual in every generation, [5, 6].

Linear ranking selection, [36], assigns a survival probability to each individual that depends only on the rank ordering of the individuals in the current population. A linear ranking of chromosomes I in the current population including N individuals

was defined by sorting the chromosomes according to increasing fitness values, $f(I)$. The rank, $rank(I)$, of the most fitted was defined to be 1 and the least fitted was defined to be N. Then a selection probability was assigned to each individual which was proportional to the individual's rank. The selection probability for chromosome I was computed by

$$p_s(I) = \frac{1}{N}\left(\mu_{max} - \frac{(\mu_{max} - \mu_{min})(rank(I) - 1)}{N - 1}\right)$$

where $\mu_{min} \in [0, 1]$ is the expected number of offspring to be allocated to the worst chromosome and $\mu_{max} = 2 - \mu_{min}$ is the expected number of offspring to be allocated to the best chromosome in the current generation.

Individuals in the new population were selected with the stochastic universal sampling algorithm developed by Baker, [37].

To minimize the variance in the number of offspring assigned to each individual, the universal stochastic sampling algorithm makes a single draw from the selection probability distribution, and uses this to determine how many offspring to assign to all parents. This procedure guarantees that the number of copies of any chromosome is bounded by the floor and by the ceiling of its expected number of copies.

As in the case of proportional selection, *elitism* has also been used to preserve chromosomes corresponding to good solution instances.

Crossover

A simple one-point crossover operation for binary coded populations have been used, [38]. Let $I = \{s_1, \ldots, s_j, \ldots, s_n\}$ and $I' = \{s'_1, \ldots, s'_j, \ldots, s'_n\}$ be two different individuals in the current population P. The crossover point was defined by randomly generating an integer j in the range [1, n]. Then the resulting crossed chromosomes are $I = \{s_1, \ldots, s_{j-1}, s'_j, \ldots, s'_n\}$ and $I' = \{s'_1, \ldots, s'_{j-1}, s_j, \ldots, s_n\}$. See Figure 5.

Mutation

Mutation was computed following a simple uniform mutation scheme for binary code populations, [26]. The integer parameter that undergo mutation, let us say s_j, is

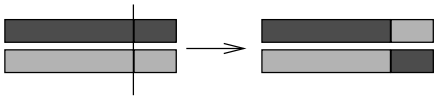


Figure 5: Crossover mechanism.

selected randomly. Then it mutates into $s'_j = 0$ if $s_j = 1$ and into $s'_j = 1$ otherwise. The mutation process is illustrated in Figure 6.

3.2.4 The Termination Condition

The algorithm stops when either the current best fitting chromosome corresponds to a solution instance that fulfills all the extra constraints defined or the number of generations reaches a given maximum threshold.

3.3 The Genetic Selector

The genetic algorithm is integrated into the constructive solver shown in Figure 2 through a *genetic selector* as illustrated in Figure 7. As required by the genetic algorithm, the input to the genetic selector includes the construction plan, the set of parameters' values and the set of extra constraints.

The genetic algorithm always returns an chromosome corresponding to the individual in the population showing the best fitness. Three different outputs from the genetic selector need to be distinguished. A possible output is an individual for which the construction plan is feasible and all the extra constraints hold. In this case an intended solution instance has been found. Notice however that this intended solution is not necessarily unique.

Another possible output is an individual for which the construction plan is feasible

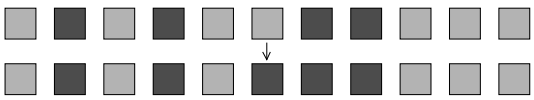


Figure 6: Mutation mechanism.

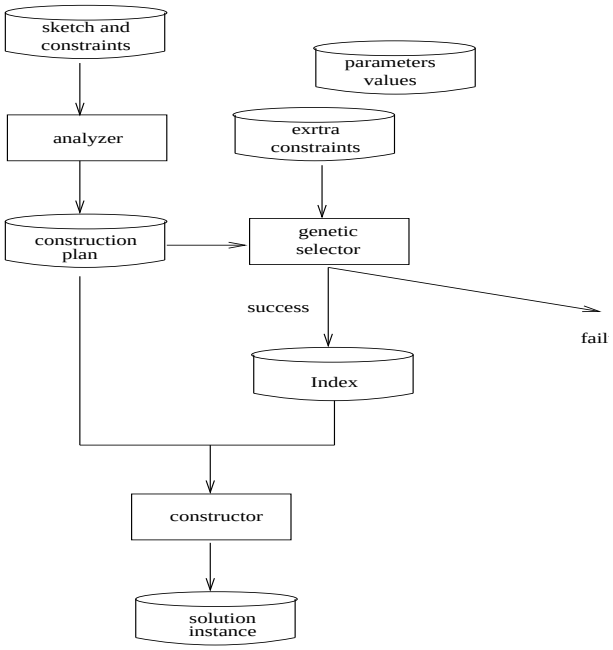


Figure 7: Integration of the genetic algorithm into the solver.

but only a subset of the extra constraints holds. In this case, a message is passed to the user interface along with the actual solution instance.

Finally when the individual does not correspond to a feasible solution, we allow the selector to fail. This information is passed to the user interface.

4 Root Multi-selection

The same set of extra constraints can characterize more than one solution instance. Therefore, the solution actually returned by the genetic algorithm and the intended solution could be different. To overcome this problem we have developed the root multi-selection technique that allows the user to request the selector to return a set of different solution instances for which the extra constraints hold.

4.1 Niching Genetic Algorithms

GAs are known to be a powerful tool for performing search in complex spaces. Anyway, one drawback they present is that when dealing with multimodal functions with peaks of unequal value, simple GAs are characterized by converging to the best peak of the space (or to a space zone containing several of the best peaks) and to lose an adequate individual sampling over other peaks in other space zones. This phenomenon represents a loss of diversity and must be avoided in those problems where one may be interested in knowing the location of other function optima.

The *niche* and *species* concepts were introduced in order to overcome this behavior [5, 39]. As the great majority of the GA concepts, they are based on translating natural notions to the field of GAs. In nature, a niche is viewed as an organism's task in the environment and a species is a collection of individuals with similar features. In this way, the formation of stable subpopulations of organisms surrounding separate niches by forcing similar individuals to share the available resources is induced.

One of the most usually employed methods for introducing niche and species in GAs is based on the *individual fitness sharing* [5, 39]. In this scheme, the population is divided in different subpopulations (species) according to the similarity of the individuals. These subpopulations form niches in two possible solution spaces: the gene and the decoded parameter ones, *genotypic* and *phenotypic sharing* respectively. Acting as in nature, the individuals belonging to each niche share the associated payoff among them. A *sharing function* is defined to determine the neighbourhood and degree of sharing for each string in the population.

4.2 The Sequential Niche Method

The root multi-selection technique is based on the *sequential niche method*, developed by Beasley *et al.* [40], to locate and maintain a set of multiple solutions.

The sequential niche method computes a set of solution instances by performing independent runs while trying to avoid the search in niches that have already been explored. In each run, the genetic algorithm obtains a solution to the problem. If the fitness of the current solution is higher than the fitness of all the solutions previously

evaluated, a new niche is stored.

To avoid searching in already explored niches, the genetic algorithm applies a penalty to the fitness of each new individual generated by the genetic operators. The penalty increases as the distance between the individual and the optima found in previous runs decreases.

The distance d_{jk} between two individuals j and k is characterized by a similarity metric. For populations like the one we have at hand where individuals are represented by bit strings, the usual distance is the well known Hamming distance, [41].

Various forms for the penalty function are possible. We have used the *power law*, [40], given by

$$G(j,k) = \begin{cases} (d_{jk}/r)^{\alpha} & \text{if } d_{jk} < r \\ 1 & \text{otherwise} \end{cases}$$

where d_{jk} is the distance between indices I_j and I_k, as determined by the distance metric. r denotes the threshold of similarity between two instance solutions, also known as the niche radius. We define it as a percentage of the maximum distance in the search domain, [42]. α is the power factor which determines the shape of the penalty function. Notice that if $\alpha = 1$ it is a linear function.

4.3 The Root Multi-selection Algorithm

The multi-selection algorithm we have implemented is given in Algorithm 2. The algorithm to handle niches is Algorithm 3 which is built by extending with the function `EvaluateP()` the basic genetic algorithm given in Algorithm 1. The input to the function `EvaluateP()` is the input to the basic genetic algorithm plus the niche radius and the power factor used to evaluate the penalty function.

5 Experimental Results

To assess the performance of the technique introduced, it has been implemented and tested. To illustrate this performance, we present a case study and briefly discuss the experimental results.

Procedure MultiSelection

INPUT

F : Functions in the construction plan.

R : Set of extra constraint.

C : Values actually assigned to the constraints.

ng : Number of generations.

ns : Number of solutions requested.

α : Power factor.

r : Niche radius.

OUTPUT

L : List of indexes of the solutions instances.

$L = \emptyset$

do

L = L + MultiGenetic(F, C, R, ng, r, α, L)

ns = ns - 1

while (ns $\neq$ 0)

return L

EndProcedure

Algorithm 2: Multi-selection algorithm.

Procedure MultiGenetic

INPUT

F : Functions in the construction plan.

C : Values actually assigned to the constraints.

R : Set of extra constraints.

ng : Number of generations.

r : Niche radius.

α : Power factor.

L : List of indexes of the solutions instances.

OUTPUT

I : Selected index.

VARIABLES

P : Population.

InitializeAtRandon(P)

EvaluateP(P, F, C, R, r, α, L)

do

I = SelectCurrentBestFitting(P)

Selection(P)

Crossover(P)

Mutation(P)

ApplyElitism(P,I)

EvaluateP(P, F, C, R, r, α, L)

ng = ng + 1

while (**not** TerminationCondition(ng, I, R))

return SelectCurrentBestFitting(P)

EndProcedure

Algorithm 3: Modified genetic algorithm.

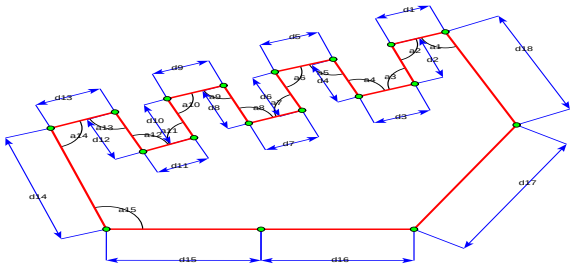


Figure 8: Geometric problem defined by constraints. Case study A.

We consider the geometric constraint problem shown in Figure 8 consisting of 18 points, 18 straight segments, 18 point-point distance constraints and 15 angle constraints. The construction plan has 16 operations where a root has to be chosen. Therefore each index included 16 binary units and the potential number of solution instances is bounded by 2^{16} = 65,536.

The intended solution instance was defined by a set including 27 extra constraints like

$$PointOnSide(P, line(P_i, P_j), side)$$

which means that point P must be placed on one of the two open half spaces defined by the straight line through points P_i, P_j, oriented from P_i to P_j. Parameter *side* takes values in $\{$*right, left*$\}$.

To check the algorithm behavior we have carried out an exhaustive evaluation of the number of extra constraints fulfilled by a number of different solution instances selected by the basic genetic algorithm. The interest was on solution instances with fitness values close to the optimal. The results are shown in Table 1. The first row shows the number of extra constraints defined to select the intended solution instance and the second row shows how many different solution instances fulfill them. Notice that the problem is really sparse.

# Extra constraints	27	26	25	24	23
# Solutions	2	0	14	24	102

Table 1: Number of extra constraints and number of different solution instances that fulfill all of them.

In the following, we discuss first the results yielded by the basic genetic algorithm, then those corresponding to the multiselection genetic algorithm.

5.1 Basic Genetic Algorithm

According to Mühlenbein, [34], five parameters are, at least, required to describe the initial state and the evolution of an artificial population of a genetic algorithm: Population size, length of the string representing individuals, initial configuration of values in the strings, mutation rate and selection law.

Investigating the behavior of the genetic algorithm with all five parameters variable would be hard to accomplish, therefore we have investigated a simpler model where only the size of the population varied. We considered populations with 25, 30, 35 and 40 individuals. The expected number of offspring to be allocated to the worst index was $\mu_{min} = 0.75$, [37] and crossover and mutation probabilities were always 0.6 and 0.2, respectively, [43].

5.1.1 The Effect of the Population

To assess the effect of the population size and the selection method on the algorithm convergence we recorded, for different population sizes, the number of extra constraints fulfilled by the individual in the population with the best fitness versus the number of generations. The experiment was conducted first applying linear ranking selection and then proportional selection. Figure 9 shows the results yielded by the algorithm for the linear ranking selection.

The algorithm convergence shows an exponential answer pattern, as expected for a natural system fed with a step input, defined by the initial population, [44]. After 30

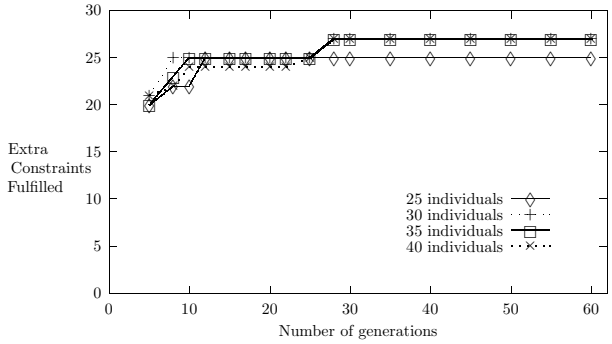


Figure 9: Linear Ranking Selection.

generations, an steady state has been always reached and when the population has 30 or more individuals, the individual selected as solution fulfills all the extra constraints.

Figure 10 shows the results yielded by the algorithm using proportional selection. Two effects can be noticed. One is that now larger populations, 35 or more individuals, are needed to select an individual which verifies all the extra constraints. The other effect is that, for the population with 40 individuals, premature convergence due to *super* individuals selected by the proportional mechanism occurred at early stages, [36, 35]. In this example, the super individual selected fulfills the set of extra constraints. However, premature convergence narrows down the search range and very often makes the algorithm to fail in finding an individual with global optimal fitness. This effect is further illustrated in Figure 11. It shows the results yielded by the algorithm fed with

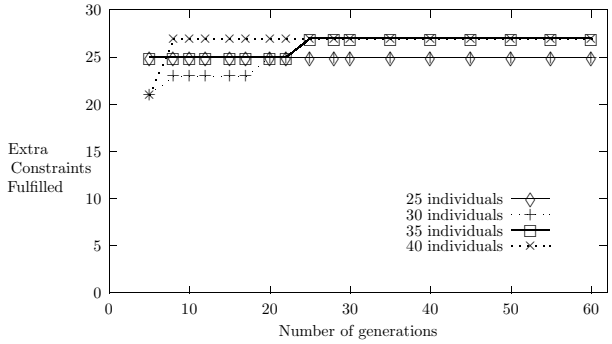


Figure 10: Proportional Selection.

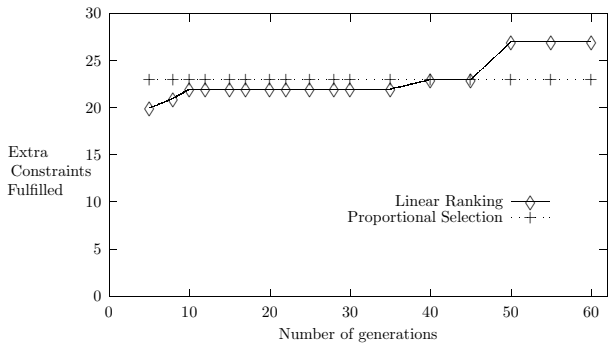


Figure 11: Premature convergence.

an initial population including 40 individuals different from those used to generate the results in Figure 10. Notice that the linear ranking selection still finds an individual with global optimal fitness.

5.1.2 Algorithm Performance

To study the performance of our basic genetic algorithm, we applied it to a number of different geometric constraint problems. Table 2 summarizes the results yielded by six different experiments selected among those reported by Luzón in [4].

The first column in Table 2 shows the number of multi valuated functions in the construction plan. The second column is the number of indices included in the pop-

n	# Indices	# Extra const.	# Sol. inst.	# Indices Eval.	%
7	12	6	2^7	61	50
10	20	12	2^{10}	110	10.7
11	30	13	2^{11}	157	7.6
16	40	27	2^{16}	561	2
18	40	39	2^{18}	534	0.3
20	50	39	2^{20}	1663	0.1

Table 2: Performance of the genetic algorithm.

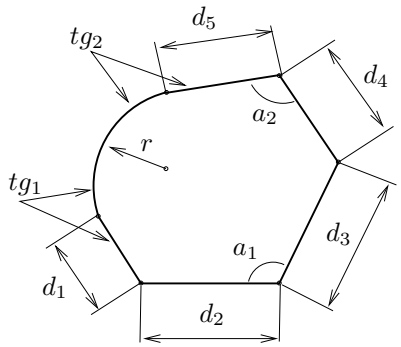


Figure 12: Geometric problem defined by constraints. Case study B.

ulation. The third column gives the number of extra constraints defined to select the intended solution. The fourth column is the number of indices in the search space. The fifth column shows the number of indices actually evaluated by the algorithm. The last column is the ratio between the figures in the fifth and fourth columns. Data in the row with $n = 16$ corresponds to the case study already considered and illustrated in Figure 8.

Data in the first row corresponds to the problem shown in Figure 12 consisting of 6 points, 5 straight segments, and a fixed radius arc of circle. The constraints were 5 point-point distances, 2 angles and 2 tangencies. The construction plan has 7 operations where a root has to be chosen. Thus each index included 7 binary units. The potential number of solution instances is bounded by 2^7 = 128.

Data in the row with $n = 20$ corresponds to the problem shown in Figure 13. The problem has 22 points and 22 straight segments. The set of constraints includes 22 point-point distances and 19 angles. The construction plan includes 20 operations where a root has to be chosen. Therefore each index included 20 binary units. The potential number of solution instances is bounded by 2^{20} and an exhaustive computation shows that only 3 of them verify the 39 extra constraints defined.

As illustrated in Table 2, the results yielded by our benchmark, [4], show that in all cases the number of indices actually evaluated by the basic genetic algorithm is a small fraction of the whole search space. Moreover, the ratio between the search space size

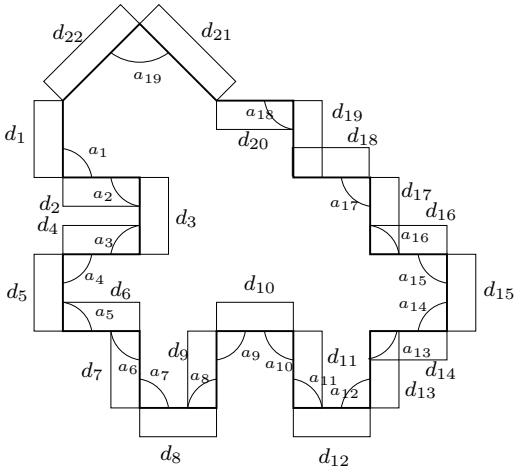


Figure 13: Geometric problem defined by constraints. Case study C.

and the number of individuals actually evaluated decreased for an increasing search space size.

The number of extra constraints fulfilled after six generations was always higher than 66%. As expected in a behavior that models a natural process, in all cases the number of extra constraints fulfilled by the indices in the current population increased exponentially until reaching an steady state. In general, about 30 generations were needed to find a solution. Therefore, the algorithm showed a great efficiency, [45].

Whenever the number of individuals in the population was equal to or greater than the number of multi valuated functions in the construction plan and the selection mechanism was the linear ranking, the individual selected at the stationary state was a global optimal, fulfilling all the required extra constraints.

5.2 Multi-selection Genetic Algorithm

To illustrate how the multi-selection algorithm works, we consider two different examples. First, consider the example in Figure 8. According to Table 1, there are only two indices which fulfill the set of extra constraints. Therefore, one can expect that when the number of solution instances required is greater than two, some of the extra

# Extra Constr. fulfilled	# Solution instances requested							
	1	2	3	4	5	6	7	8
27	1	2	2	2	2	2	2	2
25	–	–	1	1	2	3	4	5
24	–	–	–	–	–	–	–	–
23	–	–	–	1	1	1	1	1

Table 3: Performance of the multi-selection genetic algorithm. Example in Figure 8.

constraints will not hold for some solution instances selected by the multi-selection algorithm.

Recall that, in the example at hand, the number of operations where a root should be chosen is 16. The maximum distance between two solution instances occurs when all the corresponding pairs in the indices are different. Thus the maximum distance in the search space is $d_{max} = 16$. The ratio used to compute the niche radius was $r_s = 0.1$, that is, two indices are considered to belong to the same niche if they differ at most in a 10% of their components. Therefore, the niche radius was $r = r_s d_{max} \simeq 2$.

The power factor used was $\alpha = 1$, that is, the linear function. The multi-selection genetic algorithm was applied with a requested number of solutions ranging from 1 to 8. Table 3 summarizes the results.

When the number of different instance solutions requested was one or two, all the solution instances selected by the algorithm show an optimal fitness and fulfill the 27 extra constraints. As expected, when the number of requested solutions was three or more, some of the selected solutions do not fulfill all the extra constraints. For example, when requesting 4 different solution instances, two of them verify 27 extra constraints, one verifies 25 extra constraints and only 23 extra constraints hold for the last solution instance selected.

Notice that the two existing optimal solution instances are always selected. Also notice that instance solutions fulfilling only 23 extra constraints are selected whereas no solution fulfilling 24 is returned by the algorithm. A rational for this behavior is

# Extra constraints	39	38	37
# Solutions	1	0	1

Table 4: Number of extra constraints and number of different solution instances that fulfill all of them. Example in Figure 13.

that no specific search technique to try to escape from local optimal fitness is currently included in our implementation.

Now, we consider the example in Figure 13. According to Table 4, there are only one individual which fulfill the set of extra constraints. Therefore, one can expect that when the number of solution instances required is greater than one, some of the extra constraints will not hold for some solution instances selected by the multi-selection algorithm.

In this example, the number of operations where a root should be chosen is 18. The maximum distance between two solution instances occurs when all the corresponding pairs in the indices are different. Thus the maximum distance in the search space is $d_{max} = 18$. The ratio used to compute the niche radius was $r_s = 0.1$, that is, two indices are considered to belong to the same niche if they differ at most in a 10% of their components. Therefore, the niche radius was $r = r_s d_{max} \simeq 2$.

The power factor used was $\alpha = 1$, that is, the linear function. The multi-selection genetic algorithm was applied with a requested number of solutions ranging from 1 to 6. Table 5 summarizes the results.

As expected, no solution was selected such that fulfilled 38 additional constraints. When the number of requested solutions was just one, the solution instance selected by the algorithm always fulfilled all the constraints in the extra set. When the number of additional constrainrs was two or more some of the selected solutions do not fulfill all the extra constraints. For example, when requesting 5 different solution instances, one verifies 39 extra constraints, one verifies 37 extra constraints, one verifies 36 extra constraints, and two solution instances fulfill 35 extra constraints.

# Extra Constr. fulfilled	# Solution instances requested					
	1	2	3	4	5	6
39	1	1	1	1	1	1
38	–	–	–	–	–	–
37	–	1	1	1	1	1
36	–	–	–	1	1	1
35	–	–	1	1	2	3

Table 5: Performance of the multi-selection genetic algorithm. Example in Figure 13.

6 Summary and Future Work

In this paper, we have presented a new technique to efficiently search the solution space in two-dimensional constructive geometric constraint solving problems. The technique is based on a genetic algorithm which searches a solution in a potentially exponential large space of solution instances. The user defines the properties of the intended solution by adding a set of extra constraints which are used to drive the genetic algorithm in the search through the space of solution instances. The approach has been implemented on top of an already developed rule-based constructive geometric constraint solver and has been applied to a number of case studies. The results show that the technique performance is efficient and effective.

Extending the basic genetic algorithm with the sequential niche method has proved to be a convenient way to select multiple solutions requested by the user.

In ruler-and-compass geometric constraint solving, multivaluated functions have at most two different values. However, the method applies to any constructive solving technique where the construction plan is explicitly generated, all what is needed is to extend the domain where indices take values and to properly adapt the genetic operators.

Applying genetic algorithms to search the space of solution instances in constructive geometric constraint solving has shown a promising potential. To explore this

potential, we plan to further study genetic algorithms following two directions. One is to consider new types of extra constraints and the other includes to conduct experiments to identify optimum values for the population size, and crossover and mutation probabilities in geometric constraint solving.

Acknowledgments

This research has been partially supported by CICYT and FEDER under the project TIC2001-2099-C03-01.

References

[1] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. Geometric constraint solver. *Computer Aided Design*, 27(6):487–501, June 1995.

[2] L. Brisoux-Devendeville, C. Essert-Villard, and P. Schreck. Exploration of a solution space structured by finite constraints. In *ECAI 14th European Conference on Artificial Intelligence. Workshop on Modelling and Solving Problems with Constraints*, pages F:1–18, Berlin, August 2000.

[3] C. Essert-Villard, P. Schreck, and J.-F. Dufourd. Skecth-based pruning of a solution space within a formal geometric constraint solver. *Artificial Intelligence*, 124:139–159, 2000.

[4] M.V. Luzón. *Resolución de Restricciones Geométricas. Selección de la Solución Deseada*. PhD thesis, Dept. Informática, Universidad de Vigo, December 2001. Written in Spanish.

[5] D.E. Goldberg. *Genetic Algorithms in Search, Optimization , and Machine Learning*. Addison Wesley, 1989.

[6] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1996.

[7] N. Mata. Solving incidence and tangency constraints in 2D. Technical Report LSI-97-3R, Department LiSI, Universitat Politècnica de Catalunya, 1997.

[8] G. Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics*, 4(4):331–340, October 1970.

[9] L. Lovász and Y. Yemini. On generic rigidity in the plane. *SIAM Journal on Algebraic and Discrete Methods*, 3(1):91–98, March 1982.

[10] B.D. Brüderlin. *Rule-Based Geometric Modelling*. PhD thesis, Institut für Informatik der ETH Zürich, 1988.

[11] I. Fudos and C.M. Hoffmann. Correctness proof of a geometric constraint solver. *International Journal of Computational Geometry and Applications*, 6(4):405–420, 1996.

[12] R. Joan-Arinyo and A. Soto. A correct rule-based geometric constraint solver. *Computer & Graphics*, 21(5):599–609, 1997.

[13] S.C. Kleene. *Mathematical Logic*. John Wiley and Sons, New York, 1967.

[14] C. Durand. *Symbolic and Numerical Techniques for Constraint Solving*. PhD thesis, Computer Science, Purdue University, December 1998.

[15] R. Joan-Arinyo, A. Soto-Riera, S. Vila-Marta, and J. Vilaplana. Declarative characterization of a general architecture for constructive geometric constraint solvers. In D. Plemenos, editor, *The Fifth International Conference on Computer Graphics and Artificial Intelligence*, pages 63–76, Limoges, France, 14-15 May 2002. Université de Limoges.

[16] R. Joan-Arinyo and A. Soto-Riera. Combining constructive and equational geometric constraint solving techniques. *ACM Transactions on Graphics*, 18(1):35–55, January 1999.

[17] I. Fudos and C.M. Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics*, 16(2):179–216, April 1997.

[18] N. Mata. *Constructible Geometric Problems with Interval Parameters.* PhD thesis, Dept. LiSI, Universitat Politècnica de Catalunya, 2000.

[19] R. Joan-Arinyo and N. Mata. A data structure for solving geomegtric cosntraint problems with interval parameters. Technical Report LSI-00-24-R, Department LiSI, Universitat Politècnica de Catalunya, 2000.

[20] H.J. Bremermann, J. Roghson, and S. Salaff. Global properties of evolution processes. In H.H. Pattee, E.A. Edelsack, L. Fein, and A.B. Callahan, editors, *Natural Automata and Useful Simulations*, pages 3–42. Macmillan, 1966.

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

[22] T. Bäck and H.P. Schwefel. Evolution strategies i: Variants and their computational implementation. *Genetic Algorithms in Engineering and Computer Science*, pages 111–126, 1995.

[23] H.P. Schwefel. *Evolution and optimum seeking. Sixth-Generation Computer Technology Series.* John Wiley and Sons, 1995.

[24] M.J. Walsh L.J. Fogel, A.J. Owens. *Artificial intelligence through simulated evolution*. John Wiley and Sons, 1966.

[25] D.B. Fogel. *System identification trough simulated evolution. A Machine Learning approach*. Ginn Press, 1991.

[26] T. Bäck, D.B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. Institute of Physics Publishing Ltd and Oxford University Press, 1997.

[27] J.R. Koza. *Genetic Programming: On the Programming of Computer by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[28] J.R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, 1994.

[29] A.K. Mackworth. Consistency in networks of relations. *Artificila Intgelligence*, 8:99–118, 1977.

[30] A.E. Eiben and Zs. Ruttkay. Constraint-satisfaction problems. In T. Bäck, D.B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter C5.7, pages C5.7:1–C5.7:5. Institute of Physics Publishing Ltd and Oxford University Press, 1997.

[31] J.T. Richardson, M.R. Palmer, G. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In J.D. Schaffer, editor, *Third IEEE International Conference on Genetic Algorithms*, pages 191–197, San Mateo, CA, June 1989. Morgan Kauffmann.

[32] A.E. Eiben and Zs. Ruttkay. Self-adaptivity for constraint satisfaction: Learning penalty functions. In *Third IEEE World Conference on Evolutionary Computation*, pages 258–261, Nagoya, Japan, 1996. IEEE Service Center.

[33] V. Petridis, S. Kazarlis, and A. Bakirtzis. Varying quality functions in genetic algorithm constrained optimization: The cutting stock and unit commitment problems. *IEEE Transactions on Systems, Man and Cybernetics*, 28, Part B(5), 1998.

[34] H. Mühlenbein and M. Georges-Schleuter nad O. Krämer. Evolution algorithm in combinatorial optimization. *Parallel Computing*, 7:65–85, 1988.

[35] J. Grefenstette. Proportional selection and samplig algorithms. In T. Bäck, D.B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter C2.2, pages C2.2:1–C2.2:7. Institute of Physics Publishing Ltd and Oxford University Press, 1997.

[36] J. Grefenstette. Rank-based selection. In T. Bäck, D.B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter C2.4, pages C2.4:1–C2.4:6. Institute of Physics Publishing Ltd and Oxford University Press, 1997.

[37] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. *Proc. Second International Conference on Genetic Algorithms (ICGA'87)*, pages 14–21, 1987.

[38] L.B. Booker, D.B. Fogel, D. Whitley, and P.J. Angeline. Recombination. In T. Bäck, D.B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter C3.3, pages C3.3:1–C3.3:10. Institute of Physics Publishing Ltd and Oxford University Press, 1997.

[39] D.E. Goldberg K. Deb. An investigation of niche and species formation in genetic function optimization. In *Proc. of the Second International Conference on Genetic Algorithms*, pages 42–50, Hillsdale, NJ, 1989. Lawrence Erlbaum.

[40] D. Beasly, D.R. Bull, and R.R. Martin. A sequential niche technique for multimodal function optimization. *Evolutionary Computation*, 1(2):101–125, 1993.

[41] R.W. Hamming. Error detecting and error correcting codes. *Bell Systems Technical Journal*, 29:147–160, 1950.

[42] A. Pétrowski. A cleaning procedure as a niching method for generic algorithms. In *First IEEE International Conference on Evolutionary Computaion*, pages 798–803. IEEE Service Center, 1996.

[43] J.J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-16(1):122–128, 1986.

[44] K. Ogata. *State Space Analysis of Control Systems*. Prentice Hall, N.J. Englewood Cliffs, 1967.

[45] A. Eiben, P.-E. Raué, and Zs. Ruttkay. GA-easy and GA-hard constraint satisfaction problems. In M. Meyer, editor, *Constraint Processing*, LNCS Series 923, pages 267–284. Springer-Verlag, Heidelberg, 1995.