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

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

[1]Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya,
Av. Diagonal 647, 8^a, E-08028 Barcelona
{robert,tonis}@lsi.upc.es

[2]Escuela Superior de Ingeniería Informática
Universidad de Vigo,
Av. As Lagoas s/n, E-32004 Ourense
vluzon@ei.uvigo.es

May 28, 2002

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

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 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 [7] the *Root Identification Problem*.

Several approaches to solve the root identification problem have been reported in the literature. Examples are: Selectively moving the geometric elements, conducting a dialogue with the constraint solver that identifies interactively the intended solution, and preserving the topology of the sketch input by the user. For a discussion of these approaches see, for example, references [7, 9, 15, 30] and references therein.

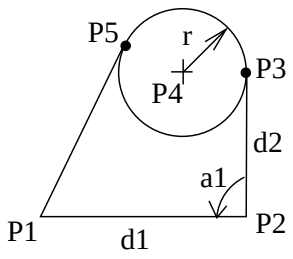
Adding extra constraints to narrow down the number of possible solutions to constraint geometric problems seems to be a simple approach. However, this approach has been carefully avoided by the field because the resulting over-constrained problem is NP hard. Moreover, the set of constraints may be contradictory, [7].

In this paper we present a new technique to solve the root identification problem by over-constraining the geometric constraint problem. The technique is based on an automatic search in the space of solutions performed by a genetic algorithm. The user specifies the intended solution instance by defining a set of additional constraints or predicates on the geometric elements which drive the search of the genetic algorithm. The approach has been implemented and the results are satisfactory, [30]. The basic technique is then extended with a *niching method* to locate and maintain multiple solutions.

The outline of the paper is as follows. In Sections 2 and 3 we briefly review the basic concepts of constructive geometric constraint solving and genetic algorithms, respectively. Section 4 is devoted to the solution instance selector based on the genetic algorithm. The niche extension is presented in Section 5. As a proof of concept, we present in Section 6 some experimental results. Finally, Section 7 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.



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.

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, [33]. 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, are determined by $2n-3$ independent relationships defined between the points, [28, 29]. 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 is well constrained, [16, 25], the formula

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

holds. By the *axiom of choice*, [10, 27], 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 show 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],

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

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 [11] and references therein for an extensive analysis of work on constraint solving. Among all the geometric constraint solving techniques, our interest focuses on the one 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.

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 [26].

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.

In general, a well constrained geometric constraint problem, [17, 25, 28], has an exponential number of solutions. For example, consider a geometric constraint

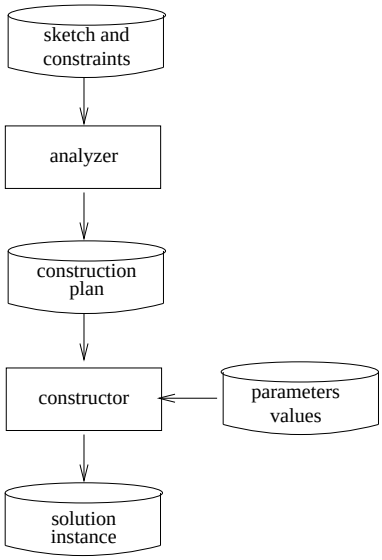


Figure 2: Basic architecture of constructive geometric constraint solvers.

$$
\begin{aligned}
P_1 &= point(0,0) \\
P_2 &= point(d_1,0) \\
\alpha_1 &= direction(P_1,P_2) \\
\alpha_2 &= adif(\alpha_1,a_1) \\
P_3 &= rc(line(P_2,\alpha_2),circle(P_2,d_2),i_1) \\
\alpha_3 &= direction(P_2,P_3) \\
\alpha_4 &= asum(\alpha_3,\pi/2) \\
Q_1 &= rc(line(P_2,\alpha_4),circle(P_2,r),i_2) \\
P_4 &= rc(line(Q_1,\alpha_3),circle(P_3,r),i_3) \\
Q_2 &= midpoint(P_1,P_4) \\
r_1 &= distance(P_1,Q_2) \\
P_5 &= cc(circle(P_4,r),circle(Q_2,r_1),i_4)
\end{aligned}
$$

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

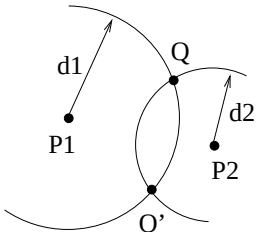


Figure 4: Possible placements of a point.

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.

Possible different 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 [15, 30].

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 [26] 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 rc and cc operation, the constructor in the solver associates an integer parameter, $i_k \in \{-1, 0, 1\}$, which identifies whether there is no solution, one or two different solutions. For details on how to compute i_k, the reader is referred to [24] and [34].

3 Basic Background on Genetic Algorithms

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, [8].

3.1 Generalities

Depending on the constructive search operations on which the algorithm is built, two families of evolutionary algorithms can be distinguished: *evolution strategies* and *genetic algorithms*. [8, 36]. An evolution strategy is a random search which uses selection and variation, [3, 37]. The *selection* operator determines those individuals in the population that survive to participate in the production of the next

population. Selection is based on the value of the fitness function, or the fitness of individual members of the population, such that members with greater fitness levels tend to survive. The variation operator, called *mutation*, introduces some sort of modification in the population members and prevents the search of the space from becoming too narrow. Evolution strategies model natural evolution by asexual reproduction with mutation and selection.

Genetic algorithms, invented by Holland, [23], are search algorithms that model sexual reproduction which is characterized by recombining two parent strings into an offspring. This recombination is called *crossover*. Crossover recombines traits of the selected individuals in the hope of producing a child with better fitness levels than its parents. Crossover is accomplished by swapping parts of strings representing two individuals in the population.

The use of genetic algorithms has been instrumental in achieving good solutions to discrete problems that have not been satisfactorily addressed by other methods, [18]. Recent surveys can be found in [1] and [18].

3.2 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 intended solution instance to a well constrained problem is specified by defining a set of additional constraints or predicates on the geometric elements. An example of extra constraint currently available to the user is defined as

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

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*}.

The resulting over-constrained problem is NP hard and one cannot expect that an effective classical deterministic search algorithm can be devised to solve it, [31]. Moreover, the set of constraints may be contradictory, [7].

Genetic algorithms have proven to be an effective technology for solving general constraint-satisfaction problems, when they are expressed as constraint optimization problems, [2, 14]. In this Section we transform the Root Identification Problem by over-constraining the geometric constraint problem into a constraint optimization problem suitable to be solved by a genetic algorithm.

Recall that we consider ruler-and-compass constructive geometric constraint solving. In this context, geometric operations correspond to quadratic equations, thus each constructive step has at most two different roots.

Let i_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 actually are feasible, that is, solution instances where no numerical incompatibilities arise in the constructor, we only

need to consider integer parameter i_j taking value in $D_j = \{0, 1\}$, where 0 stands for the first solution and 1 stands for the second different solution.

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 $I = \{i_1, \ldots, i_j, \ldots, i_n\}$ with $i_j \in D_j, 1 \leq j \leq n$. Therefore the cartesian product of sets $\mathcal{I} = D_1 \times \ldots \times D_n$ defines the space the solution instances to the geometric constraint problem belong to.

A construction plan which is 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. Clearly, the set of indexes $\{I \in \mathcal{I} \mid \Psi(I) = true\}$ is the space of feasible indexes, that is the set of indexes each defining an instance which actually is solution to the geometric constraint problem. This set of indexes is the *allowable search space*, [14].

Let Φ denote first order lolgic formula defined by 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, [14], 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.

4 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 genetic algorithms can be applied to find a solution.

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

4.1 The Genetic Algorithm

The genetic algorithm we have implemented is given in Figure 5. P is the population of indexes at the current generation. It consists on a fixed, given number of indexes in $\mathcal{I}$. The main components of the genetic learning process are described as follows.

Procedure GeneticAlgorithm
INPUT
F : Functions in the construction plan.
C : Values actually assigned to the constraints.
R : Set of extra constraints.
ng : Maximum number of generations allowed.
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

Figure 5: Genetic algorithm.

4.1.1 Initial Index Population

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 of indexes is randomly generated.

4.1.2 Index Evaluation

As stated in Section 3, 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 each constraint 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, [41], and 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 substancial knowledge on the problem at hand, [13].

To alleviate this drawback, genetic algorithms with varying fitness functions have been developed. See, for example, [13] and [39] and the references therein. Varying fitness functions make use of the natural adaptive behaviour of evolutionary algorithms based on the fact that they dynamically adjuste certain parameters according to the evolution of past experiences. As a result, locating the region where the global optimum is in the search space is favoured, [39].

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

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

where $\delta(R_i(I)) = 1$ if the solution instance associated with index I fulfills the extra constraint $R_i \in \Phi$, and $\delta(R_i(I)) = 0$ otherwise. That is, to evaluate an index fitness amounts to counting how many extra constraints its associated solution instance fulfills. MIN is the minimum fitness value in the previous generation.

As we will illustrate in Section 6, the search space is sparse because the ratio between the allowable search space and the total number of potential solutions is small. However, contrarily to what is reported in [35] and [41], the non varying fitness function did not make the genetic algorithm to fail.

Procedure RouletteWheel
INPUT
p_s : Probability distribution.
OUTPUT
c : Selected parent.

c := 1
$sum := p_s(c)$
InitializeAtRandom u in $[0, 1]$
while $sum < m$ **do**
$c := c + 1$
$sum := sum + p_s(c)$
endwhile
return c
EndProcedure

Figure 6: Roulette Wheel.

4.1.3 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, [37]. 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 solution would be too slow. Two different selection strategies were applied: Proportional selection and linear ranking selection.

Proportional selection, [19], assigns to each individual a reproductive probability that is proportional to the individual's relative fitness. If $f(I_i)$ is the fitness of index 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, [19]. See Figure 6. To preserve the best solution instance in

Procedure StochasticUniversalSampling
INPUT
p_s : Probability distribution.
L : Total number of children to be assigned.
OUTPUT
$C : (c_1, \ldots, c_N)$ where c_i is the number of children assigned to the index I_i and $\sum c_i = L$.

InitializeAtRandom u in $[0, 1/L]$
sum := 0
for $i = 1$ to N **do**
$c_i := 0$
$sum := sum + p_s(I_i)$
while $u < sum$ **do**
$c_i := c_i + 1$
$u := u + 1/L$
endwhile
endfor
return C
EndProcedure

Figure 7: Stochastic universal sampling.

each generation we applied the simplest elitism technique consisting on keeping just the best individual in every generation, [18, 35].

Linear ranking selection, [20], 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 indexes I in the current population including N individuals was defined by sorting the indexes according to increasing fitness values, $f(I)$. The rank, $rank(I)$, of the most fit was defined to be 1 and the least fit was defined to be N. Then to each individual a selection probability was assigned which was proportional to the individual's rank. The selection probability for index 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 index and $\mu_{max} = 2 - \mu_{min}$ is the expected number of offspring to be allocated to the best index in the current generation.

Indexes in the new population were selected with the stochastic universal sampling algorithm developed by Baker, [4]. See Figure 7. 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,

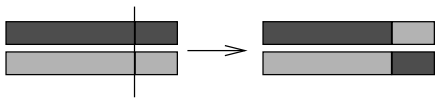


Figure 8: Crossover mechanism.

and uses this to determine how many offspring to assign to all parents. This procedure guarantees that the number of copies of any index 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 indexes corresponding to good solution instances.

Crossover

A simple one-point crossover operation for binary coded populations have been used, [6]. Let $I = \{i_1, \ldots, i_j, \ldots, i_n\}$ and $I' = \{i'_1, \ldots, i'_j, \ldots, i'_n\}$ be two different indexes 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 indexes are $I = \{i_1, \ldots, i_{j-1}, i'_j, \ldots, i'_n\}$ and $I' = \{i'_1, \ldots, i'_{j-1}, i_j, \ldots, i_n\}$. See Figure 8.

Mutation

Mutation was computed following a simple uniform mutation scheme for binary code populations, [2]. The integer parameter to undergo a mutation, let us say i_j, is selected randomly. Then it mutates into $i'_j = 0$ if $i_j = 1$ and into $i'_j = 1$ otherwise. Mutation process is illustrated in Figure 9.

4.1.4 The Termination Condition

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

4.2 The Genetic Selector

The genetic algorithm is integrated into the constructive solver showed in Figure 2 through a *genetic selector* as illustrated in Figure 10. 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.

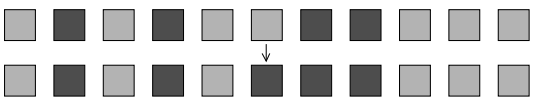


Figure 9: Mutation mechanism.

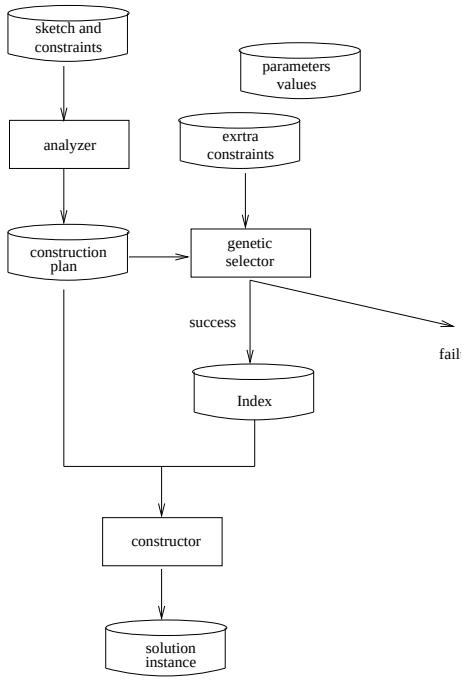


Figure 10: Integration of the genetic algorithm into the solver.

The genetic algorithm always returns an index 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 index 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 index for which the construction plan is feasible but only a subset of the extra constraints hold. In this case, a message along with the actual solution instance is passed to the user interface.

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

5 Root Multi-selection

The 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.

5.1 The Sequential Niche Method

The root multi-selection technique is based on the *sequential niche method*, developed by Beasley *et al.* [5] and by Mahfoud, [32], to locate and maintain a set of multiple solutions. A niche can be viewed as a subspace of the solution space that contains one optimal solution instance and is represented by its optimal solution.

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 niches already explored, 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 distance generally used, which is the one we have used in our experiments, is the well known Hamming distance, [22].

Various forms for the penalty function are possible. We have used the *power law*, [5], 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 indexes 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 will define it as a percentage of the maximum distance in the search domain, [40]. α is the power factor which determines the shape of the penalty function. Notice that if $\alpha = 1$ it is a linear function.

5.2 The Root Multi-selection Algorithm

The multi-selection algorithm we have implemented is given in Figure 11. The algorithm `MultiGenetic` in Figure 12 is the basic genetic algorithm shown in Figure 5 extended with the function `EvaluateP()` to handle the niches. 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.

6 Experimental Results

To assess the performance of the technique introduced, it has been implemented and tested. See [30]. To illustrate this performance, we present a case study and briefly discuss the experimental results.

Procedure MultiSeleccion
INPUT
F : Functions in the contruction plan.
R : Set of extra constraint.
C : Values actually asigned to the constraints.
ng : Maximum number of generations allowed.
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

Figure 11: Multi-selection algorithm.

We consider the geometric constraint problem shown in Figure 13 consisting of 18 points, 18 straight segments, 18 point-point distance constraints and 15 angle constraints. The potential number of solution instances is bounded by 2^{16} = 65,536. The construction plan has 16 operations where a root has to be chosen. Therefore each index included 16 binary units. The intended solution instance was defined by a set including 27 extra constraints like

$$PointOnSide(P_1, line(P_2, P_3), right)$$
$$PointOnSide(P_2, line(P_3, P_4), left)$$
$$\ldots$$
$$PointOnSide(P_{18}, line(P_1, P_2), right)$$

To check the algorithm behaviour 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 raw shows the number of extra constraints defined to select the intended solution instance and the second raw shows howmany different solution instances fulfill them. Notice that the problem is really sparse.

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

Procedure MultiGenetic
INPUT
F : Functions in the contruction plan.
C : Values actually asigned to the constraints.
R : Set of extra constraints.
ng : Maximum number of generations allowed.
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

Figure 12: Modified genetic algorithm.

# 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 them.

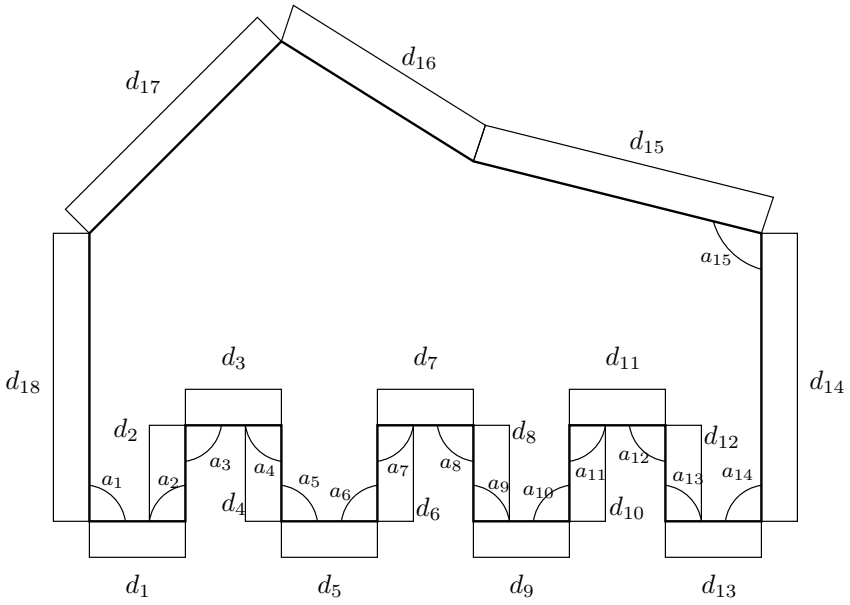


Figure 13: Geometric problem defined by constraints. Case study A.

6.1 Basic Genetic Algorithm

According to Mühlenbein, [37], 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 behaviour of the genetic algorithm with all five parameters variable would be hard to accomplish, therefore we have investigated a simpler model. The expected number of offspring to be allocated to the worst index was $\mu_{min} = 0.75$. The crossover and mutation probabilities were always 0.6 and 0.2, respectively, [21]. We considered populations with 25, 30, 35 and 40 individuals.

To assess the effect of the population size and selection method on the algorithm convergence, for different population sizes, we recorded 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 14 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 an step input, defined by the initial population, [38]. After 30 generations, an steady state has been always reached and when the population has 30 individuals or more the individual selected as solution fulfills all the extra constraints.

Figure 15 shows the results yielded by the algorithm using proportional selec-

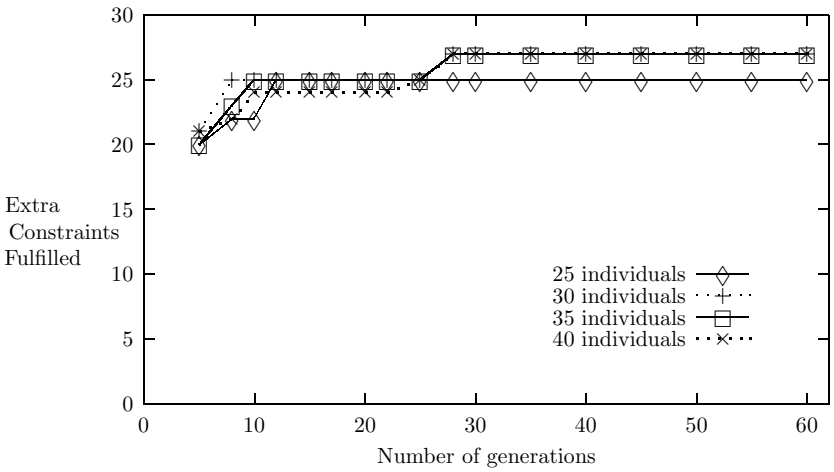


Figure 14: Linear Ranking Selection.

tion. 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 is that, for the population with 40 individuals, premature convergence due to *super* individuals selected by the proportional mechanism occurred at early stages, [20, 19]. 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 16. It shows the results yielded by the algorithm fed with an initial population including 40 individuals different from those used to generate the results in Figure 15. Notice that the linear ranking selection still finds an individual with global optimal fitness.

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 from six different experiments selected among those reported by Luzón in [30]. The first column in Table 2 shows the number of multi valuated functions in the

n	# Indexes	# Extra constraints	# Solution instances	# Indexes Evaluated	%
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.

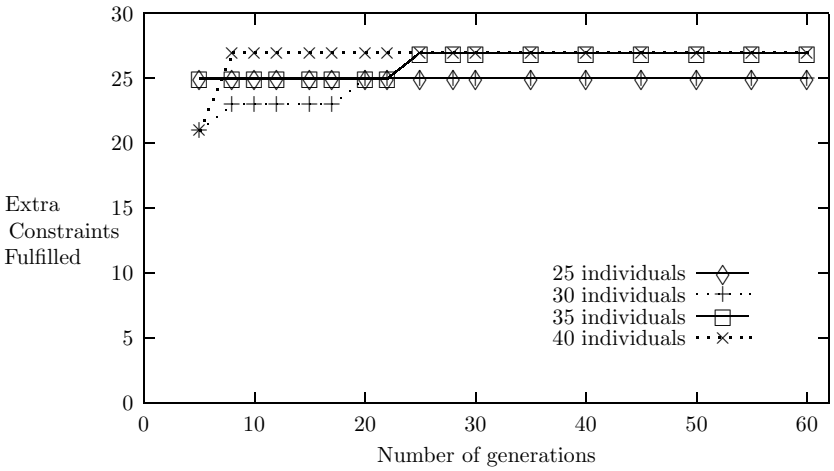


Figure 15: Proportional Selection.

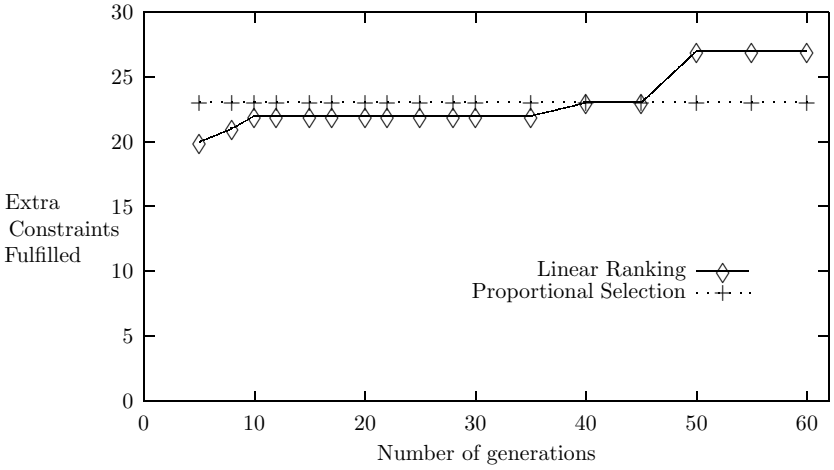


Figure 16: Premature convergence using proportional selection.

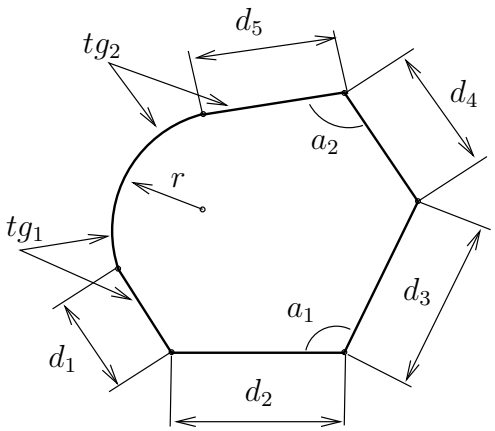


Figure 17: Geometric problem defined by constraints. Case study B.

construction plan. The second column is the number of indexes included in the population. The third column gives the number of extra constraints defined to select the intended solution. The fourth column is the number of indexes in the search space. The fifth column shows the number of indexes actually evaluated by the algorithm. The last column is the ratio between the figures in the fifth and fourth columns. Data in the raw with $n = 16$ corresponds to the case study already considered and illustrated in Figure 13.

Data in the first raw corresponds to the problem shown in Figure 17 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 raw with $n = 20$ corresponds to the problem shown in Figure 18. 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, [30], show that in all cases the number of indexes actually evaluated by the basic genetic algorithm is a small fraction of the whole search space. Moreover, the ratio between search space size and the number of individuals actually evaluated decreased for increasing search space size.

The number of extra constraints fulfilled after six generations was always higher than 66%. As expected in a behaviour that models a natural process, in all cases the number of extra constraints fulfilled by indexes in the current population increased exponentially until reaching an steady state. In general, about 30 generations were

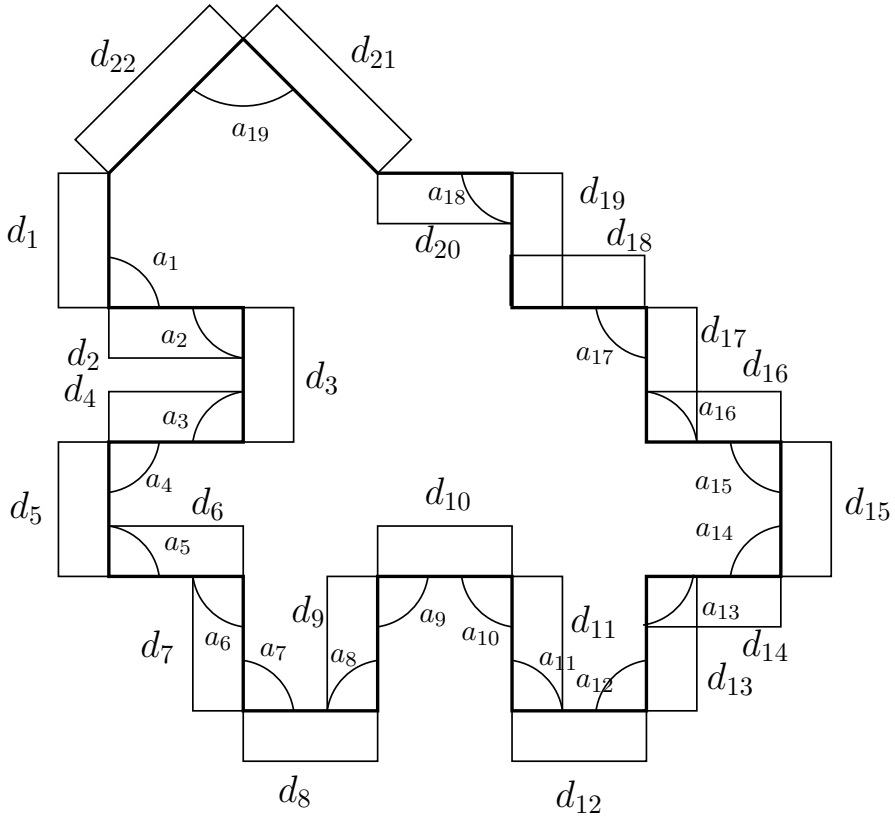


Figure 18: Geometric problem defined by constraints. Case study C.

# Extra Constraints 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.

needed to find a solution. Therefore, the algorithm showed a great efficiency, [12].

Whenever the number of individuals in the population was equal or greater than the number of multi valuated functions in the construction plan and the selection mechanism was linear ranking, the individual selected at the stationary state was a global optimal, fulfilling all the required extra constraints. This means that the algorithm effectivity is optimal, [12].

6.2 Multi-selection Genetic Algorithm

To illustrate how the multi-selection algorithm works, we consider again the example in Figure 13. According to Table 1, there are only two indexes 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 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 of indexes 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 indexes are considered to belong to the same niche if they differ at most in a 10% of their components. Therefore the niche radius was $r = r_s d_{max} = 1.6$.

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

7 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.

We have built our prototype on top of an already existing ruler-and-compass geometric constraint solver where 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 indexes 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 one includes to conduct experiments to identify optimum values for number of individuals in the population, and crossover and mutation probabilities in geometric constraint solving.

Acknowledgements

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

References

[1] In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, San Mateo, CA, 1993. Morgan Kaufmann.

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

[3] T. Bäck and H.P. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1:1–24, 1993.

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[19] 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.

[20] 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.

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

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

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

[24] 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.

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

[26] 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.

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

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

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

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

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

[32] S.W. Mahfoud. *Niching Methods for genetic Algorithms*. PhD thesis, University of Illinios at Urbana-Champain, Illinois Genetic Algorithms Laboratory, 1995.

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

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

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

[36] H. Mühlenbein. Genetic algorithms. In E. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Discrete Mathematics and Optimization, chapter 6, pages 137–171. Wiley Interscience, Chichester, 1997.

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

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

[39] 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.

[40] 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.

[41] 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.