*Computer-Aided Design and Applications*
© 2009 CAD Solutions, LLC
http://www.cadanda.com

# Search Space Pruning to Solve the Root Identification Problem in Geometric Constraint Solving

R. Joan-Arinyo[1], M.V. Luzón[2] and E. Yeguas[3]

[1]Universitat Politècnica de Catalunya, robert@lsi.upc.edu
[2]Universidad de Granada, luzon@ugr.es
[3]Universidad de Córdoba, eyeguas@uco.es

## ABSTRACT

Geometric constraint solving is at the core of parametric computer-aided geometric design where the main issue is selecting a given solution among a potentially exponential number of solution instances in the number of geometric elements involved in the geometric model. Since the user is only interested in one solution instance such that, besides fulfilling the geometric constraints, exhibits some additional properties, pruning the search space is paramount.
In this work, we report on a technique that effectively prunes the solution space search. The technique is based on making use of domain specific knowledge by storing in a dynamically maintained list solution candidates which lead to unfeasible geometric constructions.

**Keywords:** root identification problem, geometric constraint solving, pruning.
**DOI:** 10.3722/cadaps.2009.15-25

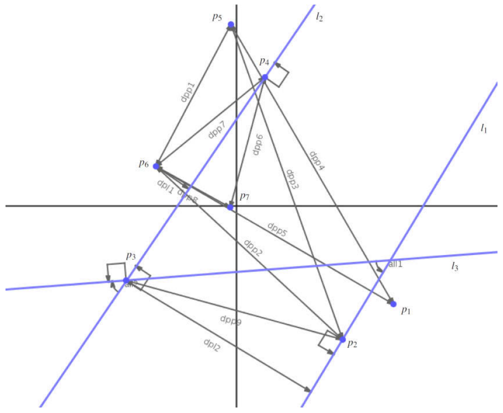
## 1. INTRODUCTION

Geometric constraint solving is arguably a core technology of computer aided design (CAD) and, by extension, of managing product design data. Since the introduction of parametric design by Pro/Engineer in the 1980s, every major CAD system has adopted geometric constraint solving into its design interface. Most prominently, 2D constraint solving has become an integral component of sketchers on which most feature-based design systems are built.
Beyond applications in CAD and manufacturing, geometric constraint solving is also applicable in other fields like virtual reality and is closely related in a technical sense to theorem proving. For solution techniques, geometric constraint solving also borrows heavily from symbolic algebraic computation and matroid theory.
The CAD field has developed sketching systems that automatically instantiate geometric objects from a rough sketch, annotated with dimensions and constraints input by the user. The sketch only has to be topologically correct where dimensions and constraints are normally not yet satisfied.
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. This solution instance is called the *intended solution*.
Selecting the intended solution amounts to selecting one solution instance among a number of different roots of a nonlinear equation or system of equations. The problem of selecting a given root was named the *Root Identification Problem* [7].

Several approaches to solve the root identification problem have been reported in the literature. Examples are: exhaustive searching the solutions space, selectively moving the geometric elements, conducting a dialogue user-system to interactively identify the intended solution, or preserving the topology of the sketch input by the user. For a discussion on these approaches see, for example, [7],[25], and references therein. Since users interact with Computer-Aided Design and Manufacturing systems in real time, and the search space is exponential, devising efficient techniques to solve the root identification problem is paramount.
Fudos *et al.*, [14], proved that the problem of finding a real solution of a system of geometric constraints solvable by a constructive method is NP-hard. From the nature of the proof, several specializations of this problem were concluded to be also NP-hard, in particular the root identification problem. Among the most common, efficient and effective techniques used to solve NP-hard optimization problems, we can find evolutionary algorithms or, in general, metaheuristics, [4].
In previous works, [21],[25], a preliminary study was conducted to asses the potential behavior of a number of metaheuristics applied to solve the root identification problem. The study showed that most of the computational effort is devoted to evaluate solution instances candidates. In this paper we present a technique to improve metaheuristics efficiency by effectively pruning the potentially exponential cardinality of the search space. The technique is based on rejecting, at the early stages of the search, solution candidates that do not lead to feasible constructions. Rejection is performed by using domain specific knowledge stored in a dynamically maintained list of known solution candidates which lead to unfeasible geometric constructions.
The remainder of this work is organized as follows. Section 2 briefly describes the main concepts involved in constructive geometric constraint solving. Section 3 formulates the root identification problem as an optimization problem. Section 4 briefly reviews the basic concepts of genetic algorithms. Section 5 defines the concepts on which pruning is performed and presents the pruning-based algorithm. To assess the efficiency of the pruning algorithm, experimental results are reported in Section 6. We close with Section 7 to draw some conclusions and to suggest 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 point-point distance, perpendicular distance from a point to a line, angle between two lines, line-circle tangency and so on. Fig. 1. left shows an example including seven points, $p_i$, $1 \le i \le 7$, and three straight lines $l_i$, $1 \le i \le 3$. The set of constraints is given on the right of Fig. 1. and includes point-point distances, *dpp()*, point-line distances, *dpl()*, angle between two lines, *all()*, and coincidences, *on()*. The goal now is to compute a placement for the geometric elements such that the constraints are fulfilled.



1. dpp($p_6$, $p_5$, *dpp1*)
2. dpp($p_6$, $p_2$, *dpp2*)
3. dpp($p_5$, $p_2$, *dpp3*)
4. dpp($p_5$, $p_1$, *dpp4*)
5. dpp($p_6$, $p_1$, *dpp5*)
6. dpp($p_4$, $p_7$, *dpp6*)
7. dpp($p_4$, $p_6$, *dpp7*)
8. dpp($p_7$, $p_6$, *dpp8*)
9. dpp($p_2$, $p_3$, *dpp9*)
10. dpl($p_6$, $l_2$, *dpl1*)
11. dpl($p_3$, $l_1$, *dpl2*)
12. all($l_3$, $l_1$, *all1*)
13. all($l_2$, $l_3$, *all2*)
14. on($p_4$, $l_2$)
15. on($p_3$, $l_2$)
16. on($p_3$, $l_3$)
17. on($p_2$, $l_1$)

Fig. 1: Geometric problem defined by constraints: (left) geometry, (right) constraints.

Many techniques have been reported in the literature that provide powerful and efficient methods for solving systems of geometric constraints. For example, see [19] 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 answer is positive, the analyzer outputs a sequence of construction steps, known as the *construction plan* that describes how to place each geometric element in such a way that all constraints are satisfied. After assigning specific values to the parameters, the constructor interprets the construction plan and builds an object instance, provided that no numerical incompatibilities arise, for example, computing the square root of a negative value.

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. Fig. 2. shows a construction plan for the object in Fig. 1., generated by the ruler-and-compass geometric constraint solver reported in [23] and available in [29]. Function names in the plan in Fig. 2. are self explanatory. For example, function *circleCR()* defines a circle by its center and radius, functions *ilc()*, *icc()* denote line-circle intersection and circle-circle intersection respectively, and so on.

1. $O$ = *pointXY*(0,0)
2. $l_1$ = *lineX*()
3. $l_3$ = *lineAP*($l_1$, *all1*, $O$, $s_8$)
4. $l_2$ = *lineAP*($l_3$, *all2*, $O$, $s_{10}$)
5. $l_1$*_PL* = *linePlD* ($l_1$, *dpl2*, $s_9$)
6. $p_6$ = *pointLD* ($l_2$, *dpl1*, $s_6$)
7. $c_2$ = *circleCR* ($p_6$, *dpp2*)
8. $p_2$ = *ilc* ($l_2$, $c_2$, $s_5$)
9. $c_1$ = *circleCR* ($p_2$, *dpp9*)
10. $p_3$ = *ilc* ($l_1$*_PL*, $c_1$, $s_7$)
11. $l_2$*_PL* = *linePlD* ($l_2$, *dpl1*, $s_6$)
12. $c_3$ = *circleCR* ($p_6$, *dpp7*)
13. $p_4$ = *ilc* ($l_2$*_PL*, $c_3$, $s_4$)
14. $c_4$ = *circleCR* ($p_6$, *dpp8*)
15. $c_5$ = *circleCR* ($p_4$, *dpp6*)
16. $p_7$ = *icc* ($c_4$, $c_5$, $s_3$)
17. $c_6$ = *circleCR* ($p_2$, *dpp3*)
18. $c_7$ = *circleCR* ($p_6$, *dpp1*)
19. $p_5$ = *icc* ($c_6$, $c_7$, $s_2$)
20. $c_8$ = *circleCR* ($p_6$, *dpp5*)
21. $c_9$ = *circleCR* ($p_5$, *dpp4*)
22. $p_1$ = *icc* ($c_8$, $c_9$, $s_1$)

Fig. 2: Construction plan for the example in Fig. 1.

A well constrained geometric constraint problem has, in general, an exponential number of solutions with respect to the number of geometric elements in the problem, [14]. 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 and that geometric operations correspond to quadratic equations. In these conditions, each constructive step has at most two different roots, that is, each point can be placed in two different locations corresponding to the intersection points of two circles. Therefore, for *n* points, once the first two points have been placed, we could have up to $2^{n-2}$ solution instances. This fact is captured in the construction plan by a set of parameters whose values allow to select a specific solution instance. For example, the construction step number 10 in the construction plan in Fig. 2. defines point $p_3$ as the intersection of a line and a circle. The specific value assigned to parameter $s_7$, called *sign*, will distinguish which intersection point is actually built out of the two possible.

Assume that $s_i$ is the sign associated with the *i*-th construction step corresponding to an equation with degree $n_i \geq 2$. To distinguish one root among the $n_i$ possible, enumerating the roots with an integer index will suffice, for example $s_i \in D_i = \{1, 2, \ldots, n_i\}$. For a more formal definition see [11].

If *L* is the total number of construction steps with more than one solution in the construction plan, we define the *index* associated with the construction plan as the ordered set of signs in the plan $I = \{s_1, s_2, \ldots, s_L\}$. Clearly, the set where the index *I* takes values is given by the cartesian product $\mathrm{S} = D_1 \times D_2 \times \cdots \times D_L$. This set defines the space where the solution instances to the geometric constraint problem belong to. If the underlying equations have at most degree two, like in the ruler-and-compass

approach, signs can take values in the set $D = \{0,1\}$ and the set where the index *I* takes values is given by $S = D^L$.
Following [7], the problem of selecting an index assignment which identifies a specific solution instance to the geometric constraint solving problem is known as the *Root Identification Problem*. This problem has been classified as NP-hard in [14]. Since geometric constraint solving is paramount in parametric solid modeling systems where user-system interactivity in real time is a must, [30], we need to devise techniques to efficiently solve the root identification problem.

## 3. ROOT IDENTIFICATION AS AN 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. Let us show that solving the root identification problem amounts to solving a general constraint-satisfaction problem expressed as a constraint optimization problem.
A construction plan which is solution to a geometric constraint problem can be seen as a function of the index $I \in S$. Moreover, the construction plan can be expressed as a first order logic formula, $\Psi(I)$, such that $\Psi(I)$ takes value true if and only if the construction plan is feasible for the index *I*, see [25]. Clearly, the set of indices $\{I \in S \mid \Psi(\mathrm{I}) = true\}$ 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,* [9].
Let $\Phi$ 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 $\Psi(\mathrm{I}) \wedge \Phi$ which has to be optimized. Then, according to Eiben and Ruttkay, [9], the triple $\langle S, f, \Psi(\mathrm{I})\rangle$ defines a *constraint optimization problem* where finding a solution means finding an index *I* in the allowable search space such that $f$ is optimal. In simple terms, the problem we are studying can be stated as follows:

> Let $A = \langle G, C, P\rangle$ be a geometric problem where $G$ is the set of geometric elements, $C$ is the set of constraints and $P$ is the set of parameters for the constraints. Let $P(I)$ be a construction plan generated by a constructive solver that defines the space of instance solutions to *A* as a function of the index *I*. Let $C'$ be the set of extra constraints that characterizes the intended solution instance. Find an index $I \in S$ such that the number of constraints in $C'$ fulfilled by the solution instance $P(I)$ is maximal.

Among the most common, efficient and effective techniques used to solve NP-hard optimization problems, we can find evolutionary algorithms or, in general, metaheuristics, [4].

## 4. BASICS ON EVOLUTIONARY ALGORITHMS

We include some basic concepts from Evolutionary Algorithms for the shake of completeness. For an in-depth study interested readers are referred to the cited works.

### 4.1 Generalities

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), [15],[20], Genetic Programming, Evolution Strategies, [2],[28] and Evolutionary Programming, [13],[12].

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, [15], 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, [15]. Recent surveys can be found in [15] and [1].

### 4.2 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 $\Phi$ 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, [8].

The genetic algorithm we have implemented is given in Fig. 3. *P* is the population of individuals at the current generation. It consists on a fixed, given number of individuals in *S*. The main components of the genetic learning process are described in Fig. 3.

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.

According to Section 3, in a given constraint optimization problem, $\langle S, f, \Psi(I)\rangle$, the function $f$, defined over $\Psi(\mathrm{I}) \wedge \Phi$, 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 $\Psi(\mathrm{I}) \wedge \Phi$.

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 $\Phi$, and $\delta(R_i(I)) = 0$ otherwise. That is, to evaluate a 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.

```
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
```

Fig. 3: Basic evolutionary algorithm.

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, [27]. The set of genetic operators we have considered includes: selection with elitism, crossover (recombination) and mutation.

*4.2.1 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, [16], and linear ranking selection, [17].

*4.2.2 Crossover*
A simple one-point crossover operation for binary coded populations have been used, [5]. 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 Fig. 4.

*4.2.3 Mutation*
Mutation was computed following a simple uniform mutation scheme for binary code populations, [1]. The integer parameter that undergoes mutation, let us say $s_j$, is 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 Fig. 5.

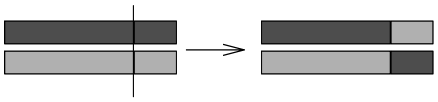


Fig. 4: Crossover mechanism.

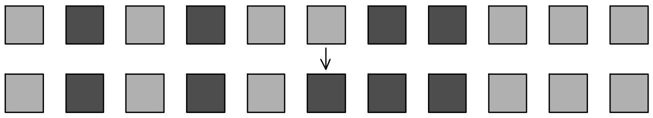


Fig. 5: Mutation mechanism.

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.

## 5. PRUNING THE SEARCH DOMAIN

In previous works, [21],[25], a preliminary study was conducted to asses the potential behavior of a number of metaheuristics applied to solve the root identification problem. The study showed that most of the computational effort is devoted to evaluate solution instances candidates. Moreover, the number of solution candidates evaluated which lead to unfeasible constructions was rather high. Therefore, metaheuristics efficiency can be improved by pruning the potentially exponential cardinality of the space search by rejecting, at the early stages of the search, solution candidates that do not lead to feasible constructions.

When searching the solution space, evolutionary algorithms systematically generate assignments for the index $I \in S$ that do not necessarily belong to the allowable space search, that is, that do lead to unfeasible constructions. The reason for a construction being unfeasible is that a construction step has been found such that the corresponding sign in the index selects a root which does not belong to the set of real numbers. In general, given a construction plan and an index assignment, there can be many unfeasible steps.

Let *P(I)* be a construction plan and $I^* = \{s_1^*, s_2^*, \cdots, s_L^*\}$ be a specific assignment to the index *I*. Recall that *I* is an ordered set. We say that $s_i^*$ is a *conflicting assignment* if the construction step associated with sign $s_i$ is unfeasible and $s_j^*$ , $1 \leq j < i$ are assignments to signs $s_j$ corresponding to feasible construction steps. Clearly, once a conflicting assignment has been found, there is no need to go further in the plan evaluation, [19],[26]. Moreover, if the conflicting assignment is $s_i^*$, index assignments that follow the pattern $I^* = \{s_1^*, s_2^*, \cdots s_i^*, s_{i+1}, \cdots s_L\}$ do not belong to the allowable search space and therefore they do not need to be explored. The subset $I_c^* = \{s_1^*, s_2^*, \cdots s_i^*\} \subseteq I^*$ is called *conflicting pattern*.

Metaheuristics are not problem specific. However they may use domain specific knowledge in the form of heuristics controlled by the upper level strategy. Nowadays more advanced metaheuristics use search experience, embodied in some form of memory, to guide the search, [4]. In our context, before evaluating the construction plan for a given index assignment, it would be useful to check whether the index assignment includes a conflicting pattern. To this aim, we record conflicting patterns $I_c^*$ in a *conflicting patterns list* which is dynamically maintained. When the size of the conflicting patterns list substantially increases, we replace conflicting patterns with the highest number of signs in the list with new shorter conflicting patterns. Notice that the higher the pattern length, the lower the number of non-constructible index assignments which are matched. Therefore the loss of efficiency is minimized. Fig. 6 shows the algorithm that evaluates a construction plan *P(I)* for a given index assignment *I* using the pruning technique presented here.

```
**Procedure** AlgorithmWithPruning
INPUT
    I: Index assignment
```

```
        P: Construction plan
        p: Parameters assignment
        CL: Conflicting patterns list
OUTPUT
        C: FEASIBLE, UNFEASIBLE
        GA: Geometry assignment
        CL: Conflicting patterns list

        # Check whether index I includes any conflicting pattern
        C := IndexInConflictingList(I, CL)
        if (C == FEASIBLE)  then
                # The index assignment does not follow any pattern in the list
                # Try to evaluate the construction plan
                EvaluateConstructionPlan(I, P, p, GA, C, I')
                if (C == UNFEASIBLE) then
                        # Construction plan evaluation has not been completed
                        # Update the conflicting patterns list
                        UpdateConflictingList(CL, I')
                endif
        endif
endprocedure
```

Fig. 6: Algorithm with space search pruning.

The conflicting patterns list is stored as a binary tree and managed with procedures IndexInConflictingList(*I*, *CL*) and UpdateConflictingList(*CL*, *I'*). We have applied the technique described in [24], working on a tree-based pattern matching approach developed in [18], and adapted to our particular context where patterns are binary strings. The tree root is a dummy node. Left subtrees store conflicting patterns starting with a 0 while patterns starting with a 1 are stored on the right subtrees. Fig. 7 shows a tree storing patterns *010, 0110, 11101* and *11110*.

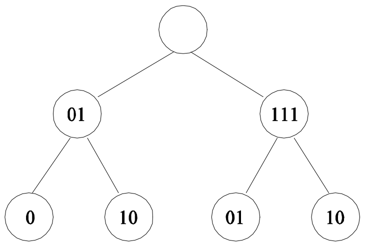


Fig. 7: Conflicting patterns list.

If pattern *110* is a new conflict pattern that must be included in the tree in Fig. 7., the resulting tree is given in Fig. 8. Assume now that the new conflicting pattern is *111*. Since it is not in the tree, it must be included in it. Moreover, since patterns *11101* and *11110* are subsumed by *111*, they must be removed from the tree. This is carried out by deleting the subtrees rooted at the node where a perfect matching with the new conflict occurs. The result is shown in Fig. 9. Then, again the patterns *110* and *111* are subsumed by *11*. Hence the corresponding subtrees could be also deleted.

EvaluateConstructionPlan(*I, P, p, GA, C, I'*) carries out the construction steps listed in the plan *P*, supplied by the solver, for the current values of *I* and *p*. If the construction succeeds, *C* returns *FEASIBLE* and *GA* stores an actual placement for the geometric elements. Otherwise, *C* returns *UNFEASIBLE* and *I'* is the conflicting pattern that yielded the construction plan unfeasible for the actual *I* and *p* assignments.

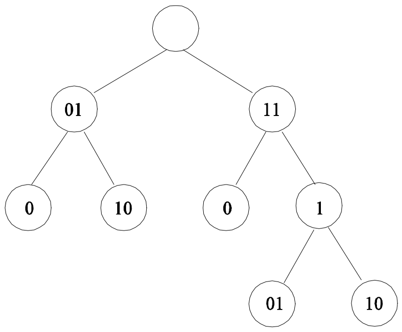


Fig. 8: Conflicting patterns list; inclusion of a new pattern.

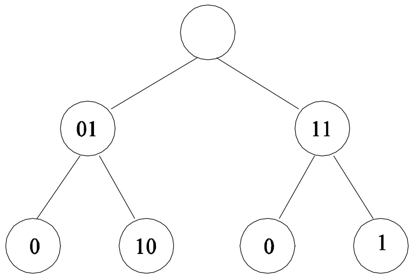


Fig. 9: Conflicting patterns list; subsumption of larger patterns.

## 6. EXPERIMENTAL RESULTS

To prove the efficiency of the proposed technique, two different metaheuristics have been selected and studied: Cross generational elitist selection Heterogeneous recombination and Cataclysmic mutation (CHC), [10], and Population-Based Incremental Learning (PBIL), [3]. For each algorithm, we developed two versions, the basic version, denoted CHC and PBIL, and the conflict list based algorithm, denoted CL_CHC and CL_PBIL. Settings for evolutionary parameters were selected according to [21] for CHC and [22] for PBIL.

A benchmark including three different problem sizes *L = {20, 25, 30}* has been set up, [31]. Using Henneberg sequences, [6], ten different problem instances have been randomly generated for each problem size. The number of runs for each algorithm and each specific problem instance was 30, each triggered with an initial random seed. The stop condition for each run was to reach at least one solution which fulfills 95% of the additional constraints defined by the user.

First we measured the number of index assignments evaluated by the algorithms. Tab. 1 summarizes the averaged values. The averaged reduction in the conflict list-based algorithms is 30% or higher measured as the ratio *(X - CL_X)/X*.

| Problem size | Algorithm | | | | |
|---|---|---|---|---|---|
| (L) | CHC | CL_CHC | | PBIL | CL_PBIL |
| 20 | 3139 | 2189 | | 8961 | 6439 |
| 25 | 4314 | 3270 | | 16973 | 12411 |
| 30 | 5815 | 4262 | | 19023 | 14445 |

Tab. 1: Number of index assignments evaluated by the algorithms.

We also assessed the expected general reduction in the search space. Notice that assuming a problem with index cardinality $L$ where the conflicting pattern $I_c^* = \{s_1^*, s_2^*, \cdots s_i^*\} \subseteq I^*$ is found, the number of index assignments that do not need to be explored is $2^{L-i}$. If the number of different conflicting patterns with cardinality *i* is $n_i$, the total index assignments that can be excluded is $n_i 2^{L-i}$. We have recorded the smallest cardinality of the conflicting patterns found by our algorithms in the experiments as well as the number of different conflicting patterns with this cardinality. Average values are given in Tab. 2. As expected, the number of different smallest conflicting patterns found depends on the technique used to find them. Although the smallest cardinalities found $|I_c^*|$ in each problem size are the same for both algorithms, in general this does not need to be the case. Just consider that they are fixed by the specific construction plan and evolutionary algorithms do not perform an exhaustive search. Pruned spaces are noticeable. For algorithm *CL_CHC* and *L = 20*, the space search cardinality is bounded by $2^{20}$ and the pruned space cardinality is bounded by $16 \times 2^5 = 2^9$.

| Algorithm | Index Cardinality (L) | | | | | |
|---|---|---|---|---|---|---|
| | 20 | | 25 | | 30 | |
| | $\|I_c^*\|$ | $\# I_c^*$ | $\|I_c^*\|$ | $\# I_c^*$ | $\|I_c^*\|$ | $\# I_c^*$ |
| CL_CHC | 5 | 16 | 7 | 35 | 8 | 91 |
| CL_PBIL | 5 | 13 | 7 | 42 | 8 | 75 |

Tab. 2: Expected general search space reduction.

## 7. CONCLUSION AND FUTURE WORKS

An algorithm to avoid the evaluation of non-constructible solutions has been proposed. The main goal is to identify conflicting patterns in the index assignment while the constructor evaluates the construction plan. Subsequent index assignments including conflicting patterns previously recorded in a conflict list dynamically maintained by the algorithm are not further considered. The efficiency of the proposed pruning technique has been proven empirically by applying it to two different metaheuristics, CHC and PBIL. Experimental results showed that pruning the search space reduces the number of construction plan evaluations in about 30%.
Currently we are exploring techniques to generate index assignments biased in such a way that they take into account the conflicting patterns currently included in the conflicting patterns list.

## 8. ACKNOWLEDGEMENTS

This research has been supported by FEDER and by CICYT of the Spanish Ministerio de Educación y Ciencia under the project TIN2007-67474-C03-01.

## 9. REFERENCES

[1] Bäck, T.; Fogel, D.-B.; Michalewicz, Z.: Handbook of Evolutionary Computation, Institute of Physics Publishing Ltd and Oxford University Press, 1997.

[2] Bäck, T.; Schwefel, H.-P.: Evolution strategies I: Variant and their computational implementation, Genetic Algorithms in Engineering and Computer Science, 111-126, 1995.

[3] Baluja, S.: Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning, Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA, 1994.

[4] Blum, C.; Roli, A.: Metaheuristics in combinatorial optimization: Overview and conceptual comparison, ACM Computer Survey, 35(3), 2003, 268-308.

[5] Broker, L.-B.; Fogel, D.-B.; Whitley, D.; Angeline, P.-J.: Recombination Handbook of Evolutionary Computation. C3.3:1-C3.3:10, Institute of Physics Publishing Ltd and Oxford University Press, 1997. T. Bäck and D. B. Fogel and Z. Michalewicz.

[6] Borcea, C.; Streinu, I.: On the number of embeddings of minimally rigid graphs, Proc. ACM Symposium on Computational Geometry SoCG'02, June 2002, 25-32.

[7] Bouma, W.; Fudos, I.; Hoffman, C.-M.; Cai, J.; Paige, R.: Geometric constraint solver, Computer-Aided Design, 27(6), 1995, 487-501.

[8] Bremermann, H.-J.; Roghson, J.; Salaff, S.: Global properties of evolution processes, Natural Automata and Useful Simulations, 3-42, Macmillan, 1996, H.H. Pattee and E.A. Edelsack and L. Fein and A. B. Callahan.
[9] Eiben, A.-E.; Ruttkay, Zs.: Constraint-satisfaction problems, Handbook of Evolutionary Computation, Pp. C5.7:1-C5.7:5, Institute of Physics Publishing Ltd and Oxford University Press, 1997.
[10] Eshelman, L.-J.: The CHC adaptative search algorithm: How to safe search when engaging in nontraditional genetic recombination, Foundations of Genetic Algorithms I, 1991, 265-283.
[11] Essert-Villard, C.; Schreck, P.; Dufourd, J.-F.: Sketch-based pruning of a solution space within a formal geometric constraint solver, Artificial Intelligence, 124, 2000, 139-159.
[12] Fogel, L.-J.; Owens, A.-J.; Walsh, M.-J.: Artificial intelligence through simulated evolution, John Wiley and Sons, 1966.
[13] Fogel, D.-B.: System identification trough simulated evolution, A Machine Learning approach. Ginn Press, 1991.
[14] Fudos, I.; Hoffmann, C.-M.: A graph-constructive approach to solving systems of geometric constraints, ACM Transactions on Graphics, 16(2), 1997, 179-216.
[15] Goldberg, D.-E.: Genetic Algorithms in Search, Optimization and Machine Learning, Addison Wesley, 1989.
[16] Grefenstette, J.: Rank-based Selection, Handbook of Evolutionary Computation, C2.4:1-C2.4:6. Institute of Physics Publishing Ltd and Oxford University Press, 1997. T. Bäck and D.B. Fogel and Z. Michalewicz.
[17] Grefenstette. J.: Proportional Selection and Samplig Algorithms, Handbook of Evolutionary Computation, C2.2:1-C2.2:7, Institute of Physics Publishing Ltd and Oxford University Press, 1997. T. Bäck and D.B. Fogel and Z. Michalewicz.
[18] Hoffmann, C.-M.; O'Donnell, M.J.: Pattern matching in trees, Journal of the ACM, 29(1), 1982, 68-95.
[19] Hoffmann, C.-M.; Joan-Arinyo, R.: A brief on constraint solving, Computer-Aided Design and Applications, 2(5), 2005, 655-663.
[20] Holland, J.-H.: Adaptation in Natural and Artificial Systems, Ann Arbor: The University of Michigan Press, 1975.
[21] Joan-Arinyo, R.; Luzón, M.-V.; Yeguas, E.: Parameter tuning for PBIL algorithm in geometric constraint solving systems, In World Congress in Computer Science, Computer Engineering and Applied Computing, International Conference on Genetics and Evolutionary Methods. 2008, 69-75.
[22] Joan-Arinyo, R.; Luzón, M.-V.; Yeguas, E.: Optimización estadística de CHC para sistemas de resolución de restricciones geométricas. In VI Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB), 2009.
[23] Joan-Arinyo, R.; Soto-Riera, A.: Combining constructive and equational geometric constraint solving techniques, ACM Transactions on Graphics, 18(1), 1999, 35-55.
[24] Lu, H.-T.; Yang, W.: A Simple Tree Pattern-Matching Algorithm, Proceedings of the Workshop on Algorithms and Theory of Computation, 2000.
[25] Luzón, M.-V.; Soto, A.; Gálvez, J.-F.; Joan-Arinyo, R.: Searching the solution space in constructive geometric constraint solving with genetic algorithms, Applied Intelligence, 22, 2005, 109-124.
[26] Mata, N.: Constructible Geometric Problems with Interval Parameters, PhD thesis, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Spain, 2000.
[27] Mühlenbein, H.; Georges-Schleuter, M.; Krämer, O.: Evolution algorithm in combinatorial optimization, Parallel Computing, 1988. Vol. 7, 65-85.
[28] Schwefe, H.-P.: Evolution and optimum seeking, Sixth-Generation Computer Technology Series, John Wiley and Sons, 1995.
[29] SolBCN, https://lafarga.cpl.upc.edu/projects/solbcn.
[30] Vila, S.: Contribution to Geometric Constraint Solving in Cooperative Engineering, PhD thesis, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona, Spain, 2003.
[31] Yeguas, E.: Root identification problem in geometric constraint solving, 2008. http://www.uco.es/~in1yeboe/benchmark.html.