# Exploration of a solution space structured by finite constraints

**Laure Brisoux-Devendeville** **Caroline Essert-Villard**
**Pascal Schreck**
Laboratoire des Sciences de l'Image, de l'Informatique,
et de la Télédétection (LSIIT, URA CNRS 1871)
Université Louis Pasteur, Boulevard Sébastien Brant
67400 Illkirch, France
{ldevende, essert, schreck}@dpt-info.u-strasbg.fr

**Abstract**

Formal CAD solvers often produce many solutions for a constraint system. It is very time-consuming to examine each of them to determine which one is the closest to the user's will. In our formal approach a construction plan is produced as a mean to represent all the solutions. In this paper, we show how a construction plan can be seen as a set of constraints of finite type. Then, we use some techniques derived from the SAT problem to efficiently explore the solution space.

**keywords:** Formal geometric constructions; Symbolic constraint solving; SAT problem; Tree pruning; Computer-aided design.

## 1 Introduction

In Computer-Aided Design (CAD), geometric constraints are used to specify rigid bodies. In order to actually handle such specified objects, different kinds of solvers have been used in CAD to compute solutions. The complete solvers are able to yield all the solutions according to the user's constraints. Unfortunately, the number of solutions is often huge, so the complete exploration can be very tedious and time consuming. We propose to use the constraint solving paradigm one more time to efficiently browse the solution space. This paper is focused on this subject: we adopt the point of view of the discrete finite constraints problems to describe and to structure the solution space.

Let us recall briefly the issues of geometric construction in CAD. When sketching an object, a draughtsman does not give an exact geometry to his drawing, but he expresses it graphically by the way of some dimensional constraints like in the elementary example of Fig. 1. Given such a dimensioned sketch, a CAD solver produces one, some or all the solutions meeting the requirements. In our example, we have two solutions given Fig. 2. Of course, the problem of solving constrained figures has been studied by many authors [44, 2, 8, 10, 11, 28, 31, 38, 39, 43] following very different approaches which we divide in two main classes: numerical approaches and formal methods. The first ones, mainly used in CAD, consist in solving numerically the equation system related to the dimensions [23, 30, 31, 6, 14]. A formal resolution of the symbolic constraints system allows to efficiently manipulate the defined figure

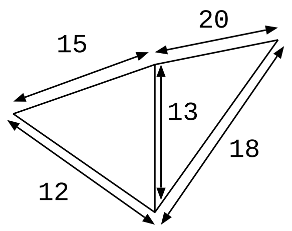

Figure 1: 2 triangles configuration: the sketch

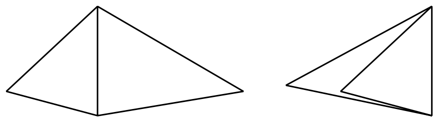
Figure 2: Two solutions

by varying parameters values [2, 10, 11, 17]. It is well known that algebraic tools for formal calculus, e.g. computer algebra systems [21], are not sufficiently efficient. On the contrary, we think that geometric formal methods are appropriate to efficiently solve geometric constraints [17]. After so many studies, the subject seems yet not topical. But, in our opinion, some aspects associated with the problem of solving geometric constraint system remain incompletely solved, such as animation of constrained figures, neutral points detection, taking into account margins of tolerance, dimension transfers, debugging of constraint systems. The “credo” of one of us (P. Schreck) is that the geometric formal approaches ensure to elegantly solve all these topics.

In this paper, we deal with another question which comes from the number of solutions of a constraint system. Actually, a constraint system does not usually define a single figure. When an infinite number of figures satisfies the constraints, the system is said *under-constrained*. When the set of solutions is finite and non empty, the system is said *well-constrained*. In the case of a well-constrained system, the exploration of the solutions space is not as easy as it seems. Indeed, the existence of polynomial equations whose degree is higher or equal to 2 from an algebraic point of view, or the existence of multiple intersections from a geometric point of view, quickly leads to a combinatorial explosion of the number of solutions.

Fig. 3 shows a sketch which is an extension of the example given by Fig. 1 and made up of fifteen adjacent triangles. Moreover, the lengths of all their sides are requested to be equal to a given dimension. In this case, we obtain 32768 solutions (triangles may be superimposed). Some of them are presented on Fig.4.

A solver is said *complete* if it is able to yield all the solutions, *incomplete* otherwise. In most cases, CAD users only want one solution figure when they design an object. Thus an incomplete solver cannot guarantee that the wanted solution is among the produced ones and it is generally difficult to find another solution. On the contrary, in the case of complete solving methods, the problem is to identify the figure expected by the user between a lot of unwanted solutions [8] and [31]. In [20], we have proposed a way to automatically select in favorable cases one solution among many. But since there are unfavorable cases, and moreover the user expectations can be fuzzy or even conflicting, we need some tools to efficiently explore the solution space. Let us note that iterative methods are usually incomplete.

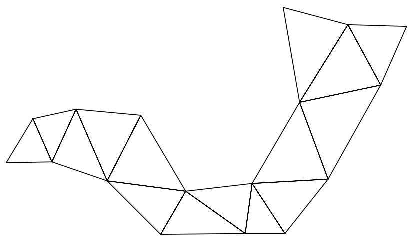

Figure 3: 15 triangles configuration: the sketch

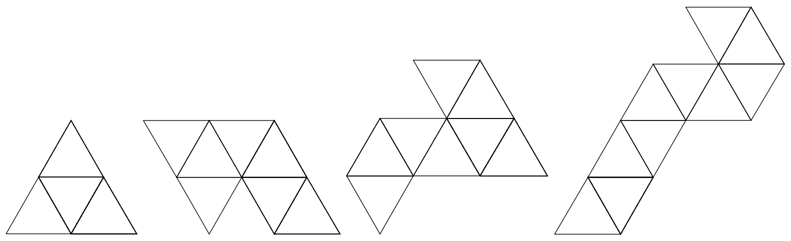

Figure 4: Four solutions among 32768 to "15 triangles"

In this paper, we show how to take advantage of our view of formal geometric resolution to propose such tools: we consider a *construction plan* as a set of finite constraints rather than an algorithmic expression. Thus each variable occurring in the construction plan must be instantiated in a finite set of numeric values and some variables depend on others. This way the solution space is not simply a set of figures but a structured space. Then we take inspiration from the SAT problem to propose some algorithms and user tools to increase the efficiency of the exploration.

The paper is organized as follows. Section 2 outlines our formal approach of constraints solving, and the relating notions. Section 3 exposes another vision of structuring the solutions space. The interaction between the membership and the dependence relations is presented in section 4 while our new heuristic is described in 5. Section 6 gives an overview of the interactive tools we propose.

## 2 Geometric constraint system solving

Although this is not the subject of this paper, we quickly present our preview works on the formal geometric construction in order to explain precisely our notion of construction plan. The original approach of our team was made a reality with the prototype called YAMS [33]. This is a formal 2D geometric solver associated with the 3D topology-based modeler *Topofil* [5]. A precise description of this association can be found in [18], so we will only present here the solver part, which acts in two stages, a symbolic one and an interpretative one.

### 2.1 Symbolic resolution

In the first stage, given a dimensioned sketch, the solver associates the geometric objects with some identifiers, then turns the constraints into formal parameters to form the geometric constraint system as it is defined below.

**Definition 1** *(Constraint system)* A *geometric constraint system* is a triple $S = (X, A, C)$, where $X$ is a set of unknowns, $A$ a set of parameters, and $C$ a set of constraints of the form $C = \{p_1(X, A), \ldots, p_m(X, A)\}$, where each $p_i(X, A)$ is a predicative term, namely a constraint, whose variables are in $X$ or in $A$.

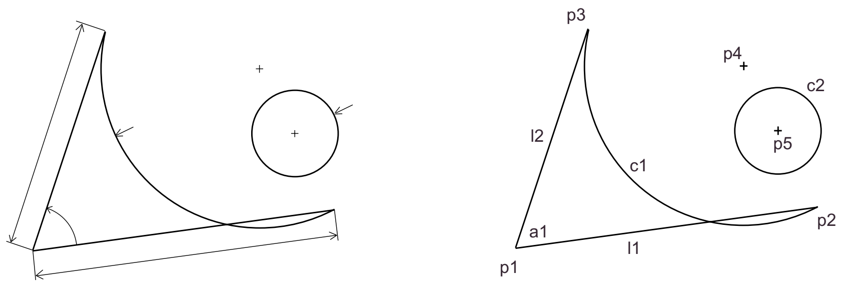

Figure 5: A sketch with constraints (left hand side) and identifiers association (right hand side)

**Example 1** An example of dimensioned sketch is shown on the left hand side of Fig.5. The first object, made up of two line segments and an arc, is subjected to topological constraints (incidence and adjacency) deduced from the sketch, and metric constraints given by the user. As shown on the diagram, there are constraints on the lengths of the two line segments, on the oriented angle between these segments and on arc and circle radii. This arc is requested to have the same center as the circle that forms the second part of the figure. The sketch as it is drawn does not respect the metric constraints, but respects incidence and adjacency constraints. The right hand side of Fig.5 shows how YAMS associates identifiers to geometric objects. By convention, $p_i$, $c_i$, $l_i$, $k_i$, and $a_i$ are the chosen names for points, circles, lines, lengths and angles, respectively. The corresponding constraint system is presented on Table 1.

Table 1: Constraints corresponding to Fig.6 example

| | | |
|---|---|---|
| `egal_p(p5, p4)` | `angle(p1, p2, p1, p3, a1)` | `onc(p2, c1)` |
| `centre(c2, p5)` | `distpp(p1, p2, k2)` | `onl(p3,l2)` |
| `centre(c1, p4)` | `distpp(p1, p3, k1)` | `onl(p2, l1)` |
| `radius(c2, k4)` | `fixorgpl(p1, l1, p2)` | `onl(p1, l2)` |
| `radius(c1, k3)` | `onc(p3, c1)` | `onl(p1, l1)` |

When the user gives a dimensioned sketch, he actually provides YAMS some numerical values, for example `distpp(p1,p2,5)`, which are abstracted on the fly to produce constraints, such as `distpp(p1,p2,k2)` and `k2 = initlength(5)`.

Then, according to the constraints, YAMS produces definitions of the form: $y := f(x_1, \ldots, x_k)$ that ensure correspondence between functional terms and identifiers. The intuitive semantic of such a definition is “when $x_1, \ldots, x_k$ are built then use function $f$ to construct $y$”, These definitions are produced as they are found to form a list of definition, called construction plan and which is the result of the formal phase.

More formally, plan $T$ can be seen as a triangular solved system, such that $S \equiv T$, that is for each instantiation $\sigma$ of the parameters of $A$, the system $S_\sigma$ has exactly the same solutions as $T_\sigma$.

**Example 2** The construction plan corresponding to the constrained sketch of Example 1 is listed in Table 2. This plan is automatically computed by YAMS and the comments are added by hand in order to make the understanding easier.

Table 2: A construction plan

```
k4 := initl(200)       - give value 200 to length k4
k3 := initl(400)
a1 := inita(1.570796)  - give value 1.570796 to angle a1
k2 := initl(300)
k1 := initl(200
p1 := initp(0,0)       - give value (0, 0) to point p1
l1 := initd(p1,0)      - line l1 is passing through p1 with slope 0
l2 := lpla(p1,l1,a1)   - l2 through point p1 with angle a1 with a line l1
c3 := mkcir(p1,k2)     - circle c3 has center p1 and radius k2
p2 := interlc(l1,c3)   - p2 is in the intersec. of line l1 and circle c3
c4 := mkcir(p1,k1)
p3 := interlc(l2,c4)
c1 := medradcir(p2,p3,k3) - circle c1 through p2 and p3 and has radius k3
p5 := centre_of(c1)
c2 := mkcir(p5,k4)
```

In addition, YAMS contains some original features resulting in a powerful 2D solver [18]. The solver is able to break the initial geometric constraint system into smaller ones. This decomposition is a bottom-up process: the subsystems are discovered during the solving process. The philosophy is to solve subfigures independently and then to glue them together with a mechanism called *assembling*. This allows to use in YAMS a collaboration of several local methods, such as knowledge-based systems and Newton-Raphson method, coordinated by a multi-agent architecture with a blackboard. An important hypothesis to ensure success is that the geometric constraint system to solve has to be *well-constrained*, that is it has a finite non-void set of solutions.

## 2.2 Interpretative stage

In the second stage, the required dimensions are used to instantiate parameters for the numerical interpretation of the construction plan. This instantiation is carried out by interpreting the definitions of the form `y := init`*x*`(`*numeric values*`)`. Since using functional terms may provide multiple results, each functional symbol is associated with a numerical *multifunction*. For example, the intersection between a line and a circle, symbolized by *interlc*, generally produces two points, and *medradcir* that builds a circle through two known points, with a known radius, generally produces two different circles. It is often useful to give a numbering to the various values produced.

**Definition 2** *(numbering)* Let $f$ be a multifunction with $n$ arguments and a maximum of $k$ results. A *numbering* of $f$ is a function $F$ with $n+1$ arguments such that

$$f(x_1,\ldots,x_n)=\{F(x_1,\ldots,x_n,1),\ldots,F(x_1,\ldots,x_n,k)\}$$

where $F(x_1,\ldots,x_n,i)\neq F(x_1,\ldots,x_n,j)$ if $i\neq j$.

Thus a definition of the form $y:=f(x_1,\ldots,x_k)$ does not simply denote an assignment but a membership relation: $y\in\{F(x_1,\ldots,x_n,1),\ldots,F(x_1,\ldots,x_n,k)\}$. From an algorithmic point of view, the existence of multifunctions in a construction plan introduces choices in the interpretation process: classically a backtracking mechanism is used when a failure occurs or when another solution is needed. This backtracking mechanism leads to considerer an *interpretation tree* such as the one in Fig. 6 on the right. Note that this way, the set of the numeric solutions of the constraint system is structured by the construction plan: the order of the definitions give the shape of the tree. The tree given in this example is complete: when two results are expected, two values are produced. It may happen that some functions do not yield the maximal number of values when a failure occurs. Thus we distinguish a potential solution tree and an effective solution tree. In case of failure, an intelligent backtracking can be used in order to prune efficiently the tree, but this mechanism is incomplete when the backtracking occurs after reaching a successful leaf.

```
k4 := initl(200)
k3 := initl(400)
a1 := inita(1.570796)
k2 := initl(300)
k1 := initl(200)
p1 := initp(0,0)
l1 := initd(p1,0)
l2 := lpla(p1,l1,a1)
c3 := mkcir(p1,k2)
p2 := interlc(l1,c3)
c4 := mkcir(p1,k1)
p3 := interlc(l2,c4)
c1 := medradcir(p2,p3,k3)
p5 := centre_of(c1)
c2 := mkcir(p5,k4)
```

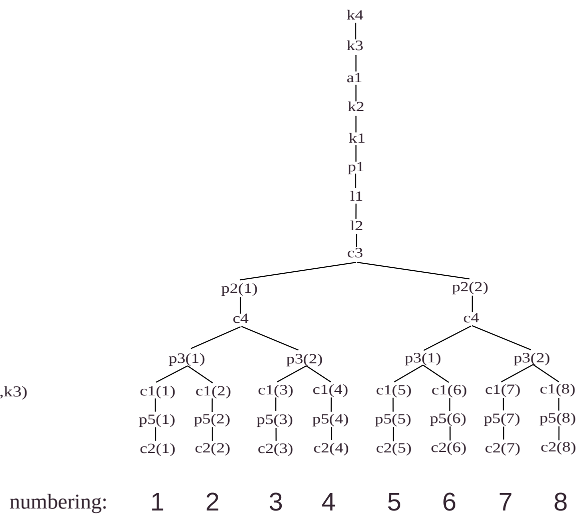
k4
k3
a1
k2
k1
p1
l1
l2
c3
p2(1)
p2(2)
c4
c4
p3(1)
p3(2)
p3(1)
p3(2)
c1(1) c1(2) c1(3) c1(4) c1(5) c1(6) c1(7) c1(8)
p5(1) p5(2) p5(3) p5(4) p5(5) p5(6) p5(7) p5(8)
c2(1) c2(2) c2(3) c2(4) c2(5) c2(6) c2(7) c2(8)
numbering: 1 2 3 4 5 6 7 8

Figure 6: Construction plan corresponding to Fig.5 and tree of solutions

**Example 3** A tree of solutions produced for our example after a parameters assignment is presented on Fig.6, at the right of the construction plan. The eight numbered solutions are shown on Fig.7. Each node corresponds to a result for an identifier, which is written with the result number in brackets. Note that the results of $p3$ in the subtree leading to solutions $\#5$ to $\#8$ are labeled the same way as in the subtree leading to solutions $\#1$ to $\#4$ because $p3$ does not depend on $p2$, so they really are identical. We show below how to exploit this feature.

A construction plan is a *list* of definitions whose order is given by the solver, but not any list of definitions can be interpreted as a construction plan. Indeed, any variable

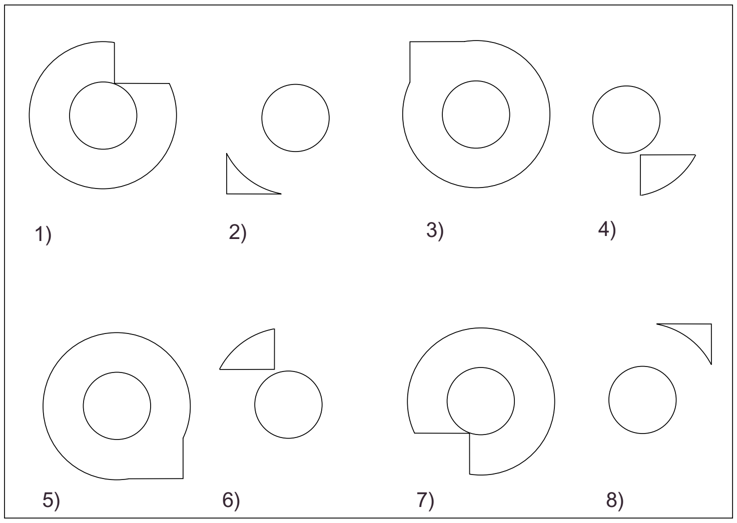

Figure 7: The generated solutions

must appear one and once as a left member of a definition and when it does in the definition $y = f(x_1, \dots, x_k)$, all the variables $x_i$ must have been defined before.

## 3 Finite constraints and structured solution space

The previous section explains how the algorithmic aspect of a construction plan with multifunctions naturally leads to a structuration of the solution space. One could think that this structuring is artificial and depends closely on the *way* the constraint system is solved rather than on the system itself. It is not really true. Consider a set $E$ of numeric solutions of a given numeric constraint system $S = (C, X, \emptyset)$. $E$ is a set of tuples $(e_1, e_2, \dots, e_n)$ whose components are values for the unknowns of $X$ in order to satisfy $C$. For each unknown $x_i$, there is a number $n_i$ of possible values corresponding to the $i$th component. Suppose we sort the unknowns according to the number of possible solutions in order to have a tuple solution where $n_i \leq n_j$ if $i < j$. Note that there is only one solution for the first unknown i.e $n_1 = 1$, thus all the tuples have the same first component value. Then the set of solutions can be organized in a tree structure by factorizing the common values: level $i$ corresponding to the values of component $i$, so each branch of the tree is a tuple of the set of solutions. If we do not take into account the descendants order, there is one tree structure for each kind of sorting of the unknowns, according to the number of possible solutions. One of these trees corresponds to the interpretation tree provided by our construction plan.

This fact suggests to consider a construction plan from another point of view. This is done by replacing definition $y = f(x_1, \dots, x_k)$ by the conjunction of constraints $y \in \{t_1, t_2, \dots, t_m\}$ and $y \to x_1, \dots, y \to x_k$, in which $t_i = F(x_1, \dots, x_k, i)$ and $y \to x_1$ means $y$ depend on the variable $x_1$. As a consequence, the definition producing $x_1$ must be evaluated before the one producing $y$. Another type of constraint can be considered namely $x << y$ which means that the definition producing $x$ is evaluated before the one producing $y$. We have $y \to x \supset x << y$, but the opposite is false. The membership relation with the effective order $<<$ relation corresponds exactly to a construction plan

Table 3: Combinatorial complexity of trees

| **First plan** | | **Second plan** | |
|---|---|---|---|
| `k1 := initl(200)` | `-- 1 node` | `k1 := initl(200)` | `-- 1 node` |
| `k2 := initl(250)` | `-- 1 node` | `k2 := initl(250)` | `-- 1 node` |
| `k3 := initl(300)` | `-- 1 node` | `k3 := initl(300)` | `-- 1 node` |
| `a1 := inita(0.7)` | `-- 1 node` | `a1 := inita(0.7)` | `-- 1 node` |
| `a2 := inita(0.5)` | `-- 1 node` | `a2 := inita(0.5)` | `-- 1 node` |
| `p1 := initp(0, 0)` | `-- 1 node` | `p1 := initp(0, 0)` | `-- 1 node` |
| `l3 := initd(p1, 0)` | `-- 1 node` | `l3 := initd(p1, 0)` | `-- 1 node` |
| `c2 := mkcir(p1, k3)` | `-- 1 node` | `c3 := mkcir(p1, k1)` | `-- 1 node` |
| `p3 := interlc(l1, c2)` | `-- 2 nodes` | `c2 := mkcir(p1, k3)` | `-- 1 node` |
| `c3 := mkcir(p1, k1)` | `-- 2 nodes` | `p3 := interlc(l1, c2)` | `-- 2 nodes` |
| `c4 := mkcir(p3, k2)` | `-- 2 nodes` | `l4 := lpla(p1, l3, a1)` | `-- 2 nodes` |
| `p2 := intercc(c3, c4)` | `-- 4 nodes` | `l5 := lpla(p3, l3, a2)` | `-- 2 nodes` |
| `l2 := line(p1, p2)` | `-- 4 nodes` | `p4 := interll(l4, l5)` | `-- 2 nodes` |
| `l1 := line(p3, p2)` | `-- 4 nodes` | `c4 := mkcir(p3, k2)` | `-- 2 nodes` |
| `c1 := mkcins(l1, l2, l3)` | `-- 16 nodes` | `p2 := intercc(c3, c4)` | `-- 4 nodes` |
| `l4 := lpla(p1, l3, a1)` | `-- 16 nodes` | `l2 := line(p1, p2)` | `-- 4 nodes` |
| `l5 := lpla(p3, l3, a2)` | `-- 16 nodes` | `l1 := line(p3, p2)` | `-- 4 nodes` |
| `p4 := interll(l4, l5)` | `-- 16 nodes` | `c1 := mkcins(l1, l2, l3)` | `-- 16 nodes` |

whereas the membership relation with the dependence order corresponds to all possible interpretation orders.

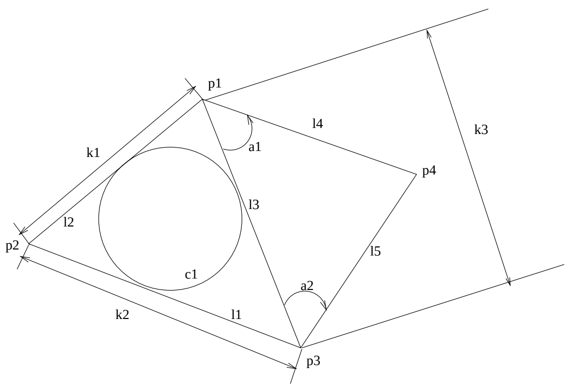

Figure 8: Sample example

We take advantage of this fact to find an order $<<$ compatible with $\rightarrow$ and more efficient from the user's point of view. Let us take the small naive example of dimensioned sketch on Fig. 8 and consider then two construction plans solving this dimensioned sketch given on Table 3. The maximum number of nodes per level is indicated, and `mkcins` refers to the function building the four circles (in general) tangent to three lines. The number of solutions is equal to the number of leaves and is, of course, the same for the two plans, but the total maximal number of nodes are respectively of 90 and 47. This fact is not important when only one branch is selected like in [20], but it becomes interesting when a naive complete exploration is done. Moreover, it points out the interdependence between the membership, $\rightarrow$ and $<<$ relations (Note that in our example, only 44 calculi are needed).

The three kinds of finite constraints described above are naturally associated with abstract data types. Thus the dependence relation ($\rightarrow$) corresponding to a well-formed construction plan is a partial order over the variables: it can be represented by a DAG that we call the *dependence graph*. The notion of multifonction is classically captured by a function returning a list of values. In this case all the results are computed at the same time: the advantage is that the global computation is often lighter than repeating $n$ times one computation, the drawback is that some unneeded values can be computed for nothing. This list can also be filled when needed with the opposite advantages and drawbacks. The membership relation is then represented as the membership in a list. In the end, the previous relation $<<$ exactly corresponds to a construction plan, i.e. a *list of definitions* which are all well-founded: in a definition, the arguments of the functional term must be variables defined in previous definitions. The algorithmic relation between $\rightarrow$ and $<<$ is that the order of the defined variables in a construction plan is the result of a topological sorting over the dependence graph.

## 4 Membership and dependence relations

In finite constraints resolution, an intelligent backtracking is often used after a failure. Let's briefly describe it. Usually, naive backtracking because of a failure during the interpretation of definition $y = f(x_1, \ldots, x_k)$, is given by choosing the next solution of definition $y' = g(z_1, \ldots, z_l)$, which is immediately over $y$. If $y'$ is not an argument of $y$, computing again $y = f(x_1, \ldots, x_k)$ leaves $y$ still unchanged and leads to a new failure.

In intelligent backtracking, the nearest argument of $y$ is selected to continue interpretation [32]. To go to the nearest argument it is necessary to use $\rightarrow$ relation. This method can reduce the search by pruning interpretation tree during a *real failure*. Obviously, this method is not complete when it is used to compute all solutions. The next algorithm allows complete exploration while the number of calculi is minimized.

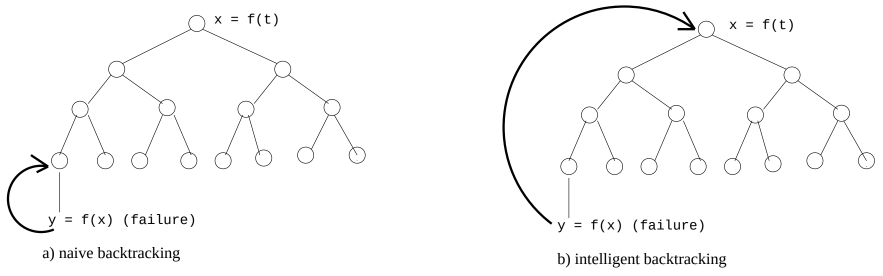

Figure 9: Intelligent backtracking

With a construction plan, we of course have the logic:

$$\begin{gathered} y \rightarrow x_1 \wedge \ldots \wedge y \rightarrow x_k \wedge x_1 := v_1 wedge \ldots x_k := v_k \\ \supset \\ y \in \{F(v_1, \ldots, v_k, 1), \ldots F(v_1, \ldots, v_k, m)\} \end{gathered}$$

that means that when arguments are not modified, the set $f(x_1, \ldots, x_k)$ have not to be computed again. This is patently obvious, classical backtracking has no memory, all solutions of the cut branch are forgotten.

The method used is inspired by the Unix "make" program. Definitions are dated and, during a computation, if the date of all arguments are older than the current definition, solutions are not computed again but stored solutions are used.

All solutions coming from computing multifunction are stored: either once if the computing multifunction returns all solutions, or progressively when they are computed as needed.

Each definition needs to have all computed solutions. For this, each definition is associated with the list of produced values and a mark on the current value: during a backtracking on this definition, if the next solution exists this mark is updated, otherwise *fail* is returned.

During the computation of values of a definition, the dates of all arguments are verified. If one of them is newer than the current definition, the list of solutions is removed and the multifunction is computed with new values. Else the mark of current solution is put back to the first solution. Algorithm 1 sums up this:

**Algorithm 1** Procedure compute_solutions($D$)

```
Require: $D$ a definition
Ensure: list of solutions of $D$, if they exist
  out ← empty
  compute ← false
  if sol not exists then
    compute ← true
  else
    for each definition arguments $a$ of $D$ do
      if when_compute_sol($a$) $>$ when_compute_sol($D$) then
        compute ← true
      end if
    end for
  end if
  if compute then
    out ← compute_sol($D$)
    update_when_compute_sol($D$)
  end if
  return out
```

- Procedure `when_compute_sol(`$a$`)` returns the date of definition $a$,
- Procedure `update_when_compute_sol(`$D$`)` updates the date of definition $D$.
- Procedure `compute_sol(`$D$`)` computes values of definition $D$, using the value of its definition arguments and its multifunction.

With the worst construction plan given on Table 3 there is no failure, only 44 calculi are computed, instead of 90 foreseen: values for `l4`, `l5` and `p4` are computed one time (3 calculi) despite of 16 branches in the associated search tree to this construction plan (48 calculi).

Even with a naive backtracking, this method allows to compute only necessary calculi. But, in case of real failure, part of the search tree will be explored again even if there is no solution. Fortunately, this method can be jointly used with intelligent backtracking.

This method could be extended by storing more levels. That would take more place in memory but be faster. This new method has not been tested yet.

## 5 Learning from failure

As we said previously, the size of the tree depends on the order of the definitions. To reduce the number of nodes, two classes of heuristics can be distinguished: static heuristics and dynamic heuristics.

Static heuristics can consist in reorganizing the definitions of the construction plan using a topological sorting with respect to the dependence graph.

We study here a dynamic heuristic inspired from SAT technics. Let us first recall the formal framework under consideration.

### 5.1 Going over SAT

SAT consists in checking the satisfiability of a boolean formula in conjunctive normal form (CNF). A CNF formula is a set (interpreted as a conjunction) of clauses, where a clause is a disjunction of literals. A literal is a positive or negated propositional variable.

An interpretation of a boolean formula is an assignment of truth values to its variables. A model is an interpretation that satisfies the formula. Accordingly, SAT consists in finding a model of a CNF formula when such a model does exist or in proving that such a model does not exist. SAT is an NP-complete problem meaning that all algorithms to solve it should be exponential in the worst cases, unless $P = NP$ [12].

However, not all SAT instances do exhibit the same difficulty, with respect to usual algorithms to solve them.

Recently, very simple stochastic search techniques have proved their efficiency in solving large and hard consistent SAT problems (see e.g. [42, 41] [3] [27] [35]). However, such techniques are logically incomplete since they do not cover the whole search space. Accordingly, they cannot be used as such to prove the inconsistency of SAT instances (see however [34]). Actually, the most efficient logically complete techniques are still based on the classical Davis Logemann Loveland's procedure called DPLL.

Some new efficient versions of DPLL have been proposed recently, extending its practical scope to a really significant extent. Among them, let us simply mention the most efficient ones, namely C-SAT [7] [15], Tableau [13], POSIT [22], Satz [29], Relsat [4], GRASP [32] and DP+TSAT [34, 36].

DPLL leads a binary search tree, whose vertices are labeled by a literal representing their assignment in the partial interpretation, and whose nodes represent the subset of clauses which have not been satisfied by the partial interpretation.

One of the key feature for efficiency in the DPLL procedure lies in its branching strategy. Accordingly, many branching rules have been proposed in the literature ([26], [7], [22]).

A new branching rule has been proposed by Brisoux *et al.* [9], inspired from Morris's work [37], and can be described as follows: whenever some clauses have been shown unsatisfiable at some steps of the search process, this information should not be neglected in the remaining search process. On the contrary, in the development of the other branches of the search tree it could be efficient to try to encounter these situations of unsatisfiability again, everywhere and as soon as possible, modulo the other factors allowing an efficient branching rule to be obtained. This can be done by selecting with a higher priority the literals occurring in these clauses. A similar method had been proposed by Abreu *et al* [1] for improving the execution of non-deterministic concurrent logic languages.

This idea can be simply realized by adding a weight $\beta_c$ to each clause. Each time DPLL selects a propositional variable that would immediately lead to inconsistency, each initial clause $c$ of the SAT instance that would be shown unsatisfiable at this step of the search tree has its factor $\beta_c$ increased by a given value $\gamma$, and its importance is thus increased in the further search.

## 5.2 Description

**Algorithm 2** Procedure Interpretation($G$)

```
Require: $G$ a construction plan
Ensure: display solutions of $G$, if they exist
  def $\leftarrow$ head($G$)
  first_loop $\leftarrow$ true
  while stack_is_not_empty() or first_loop do
    while def exists do
      first_loop $\leftarrow$ false
      out $\leftarrow$ compute_solutions(@def)
      if out = NO_SOLUTION then
        increase_weight(def)
        reorganize(def,$G$)
        def $\leftarrow$ pop_after_reorganized()
      else
        def.position_solution $\leftarrow$ next(def.position_solution)
        if def.position_solution not exists then
          def $\leftarrow$ pop()
        end if
        add_to_computed_solution(def.position_solution)
        if out = ONE_SOLUTION then
          def $\leftarrow$ next_in($G$)
        else
          if out = MORE_ONE_SOLUTIONS then
            push(def)
            def $\leftarrow$ next_in($G$)
          else
            def $\leftarrow$ next_in($G$) (*out = MORE_ONE_SOLUTIONS_ALREADY_POP*)
          end if
        end if
      end if
    end while
    display_computed_solution()
  end while
```

We extend this heuristic to our problem by introducing a weight to each definition. We propose to reorganize the definitions by taking into account this new feature.

The motivation behind this heuristic is the same as in the SAT problem: to favour definitions that have lead to a failure at some previous steps of the search process, without the need of recording anything else to reduce the search tree.

At the beginning, the weight of all definitions is set to a constant $c$, as in SAT. When the function of the definition has no solution (this is a failure), its weight is increased by one and the plan is reorganized taking into account this weight and the positions of its argument definitions because this definition still verifies $\rightarrow$. This procedure is described more formally in Algorithm 2.

Procedure `compute_solutions(@def)` (where the sign @ stands for the address of variable) updates `def.solutions` where solutions are stored. The position of the current solution is indicated by `def.position_solution`. With its returned values, it can be decided to reorganize the construction plan, by pushing/poping on a stack of definitions (to simulate recursivity). Function `pop_after_reorganized()` pops all values stored in the stack, between the old position of `def` and the new one. The top of the resulting stack is returned.

Function `increase_weight` increases the weight of a definition by one, and function `reorganize (def,`$G$`)` tries to find the best place in the construction plan such that all definitions are classified by decreasing weight, as much as possible, and that $\rightarrow$ is respected.

Let *def* be the definition to be moved up in construction plan $G$. If the predecessor of *def* in $G$ is not an argument of *def* and if its weight is lower, then definition *def* can be inserted before it. This procedure can be repeated until a good place is found. That gives the following procedure:

**Algorithm 3** Procedure reorganize(*def*,$G$)

```
Require: def a definition, G its construction graph
Ensure: G reorganized taking into account the weight
  current ← def
  here ← current
  pred ← current
  exit ← false
  while not exit and (weight(current) > weight(pred)) and no_top_of(G) do
    if respect_NADS(current) then
      pred ← current
      current ← before(current)
    else
      here ← pred
      exit ← true
    end if
  end while
  if here ≠ def then
    insert_at(def,here)
  end if
```

- Procedure `weight(`$d$`)` returns the weight of definition $d$,
- Procedure `respect_NADS(`*def*`)` returns true if it respects relation $\rightarrow$, when definition *def* is moved,
- Procedure `insert_at(def,here)` inserts definition *def* after definition *here* in the construction graph.

The new YAMS procedure allows dynamical reorganization. After computing all solutions, the order of definitions is almost optimal if there had been enough failures.

## 6 User management of constraints

The algorithms used in previous sections are transparent for the user: the finite constraints are handled by the software in order to cut down the number of calculi during the complete exploration of the interpretation tree. Despite these efforts, it remains often many solutions to be examined by the user due to the sort of constraints, and this task quickly becomes tedious even if each solution is efficiently computed. We propose here some tools which can help the user to progressively select a few solutions. These tools are not yet implemented. The main idea is that the user can manage the finite constraints with the help of intuitive high level operations.

For instance, the control of the interpretation order can be given to the user. Since the dependence relation cannot be broken, the algorithm which reorganizes the construction plan using the weight of definitions can be used. Thus the user can change the priority of some parts of the figure in order to speed up the backtracking of this part or in the contrary, to freeze another part by increasing the weight of some definitions. This way the user can affect smoothly relation $<<$ which gives the order of interpretation.

The user can more radically act on the interpretation process without changing relation $<<$. A proposed tool consists in grouping definitions in order to make a *block of definitions* whose behaviour will be the same as a plain definition:

$$fig := block(def_1, def_2, ..., def_m)$$

where $fig$ is a tuple made up of the variables defined by $def_1, \ldots, def_m$. Thus, the entire block is moved when its relative weight changes, the value of the whole interpretation of the block is memorized and a freezing of the block is easier done. This method is based on the optimization of the membership relation by storing a large part of it. For the user, the main advantage is to be able to freeze a whole subfigure (numerically or syntaxically by numbering a branch of this partial subtree) or to limit the backtracking to this block. This approach is favoured by our decomposition method which leads to consider *subfigures* as independent of the rest of the figure to some extend. Thus, natural blocks exist in the construction plan.

Another approach seems very promising: regarding the list of definitions as a list of clauses, we plan to use tools inspired from Prolog debugger. With these tools, the user can execute step by step the construction plan, force the examination of another value given by a multifunction, put some spy points, and so on. These tools take into account membership and relations $<<$. Moreover, with our data structure, we can take advantage of the dependence relation by allowing the user to return back to the arguments of a given definition and then to backtrack from this node. There are many such tools and we have to embed them it with some user-friendly graphic interface, but we have not studied this problem yet. The graphical nature of our problem is well suited in order to develop some intuitive tools for exploring a wide-ranging finite set.

## 7 Conclusion

Geometric formal resolution is well suited for parametric design in CAD. In this paper, we present some technics to take advantage of this approach: the construction plan is managed as a set of finite constraints in order to speed up its interpretation and to give to the user some exploration tools.

The “first fail” approach is used by means of coming from the SAT problem: some other heuristics could be adapted later. Note that this work is still in progress. Some experiments have to be done to adjust the parameters of our heuristics. And the user-friendly tools are still to be developed.

Nevertheless, we think that the technics used in CSP in general can be fruitfully used in this domain and in return, the tools developed in this framework can be used in other domains.

## References

[1] S. Abreu, L. M. Pereira and P. Codognet. Improving Backward Execution in the Andorra Family of Languages. In *Proceedings of the Joint International Conference and Symposium on Logic Programming, (JICSLP’92)*, pages 384–398, November, 1992.

[2] B. Aldefeld, H. Malberg, H. Richter and K. Voss. Rule-based variational geometry in computer-Aided Design. *Artificial Intelligence in Design*, D.T. Pham editor, Springer-Verlag, 1991.

[3] *Artificial Intelligence*, volume 81, March 1996. Special volume on frontiers in problem solving: Phase Transition and Complexity.

[4] R. J. Bayardo Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI’97)*, pages 203–208, Providence, Rhode Island, USA, 27-31 July 1997.

[5] Y. Bertrand, J.F. Dufourd. Algebraic specification of a 3D-modeller based on hypermaps. *Computer Vision - GMIP*, 56(1):29-60, 1994.

[6] A. Borning and B. N. Freeman-Benson. The OTI Constraint Solver: A Constraint Library for Constructing Interactive Graphical User Interfaces. In *Principles and Practice of Constraint Programming - CP’95, First International Conference*, pages 624–628, 1995.

[7] Y. Boufkhad. *Aspects probabilistes et algorithmiques du problème de satisfiabilité*. Ph.D. Thesis, Université de Paris VI, 1996.

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

[9] L. Brisoux, É. Grégoire, and L. Saïs. Improving Backtrack Search for SAT by Means of Redundancy. In Z W. Ras and A Skowron, editors, *Proceedings of the Foundations of Intelligent Systems, Eleventh International Symposium on Methodologies for Intelligent Systems, ISMIS’99*, volume 1609, pages 301–309, Warsaw, Poland, 8-11 June 1999. LNCS, Springer.

[10] B. Brüderlin. Using Prolog for constructing geometric objects defined by constraints. In *Proceedings of EUROCAL’85*, LNCS 204, Springer-Verlag, pages 448-459, 1985.

[11] B. Brüderlin. Automatizing geometric proofs and constructions. In *Proceedings of Computational Geometry '88*, LNCS 333, Springer-Verlag, Berlin, pages 232-252, 1988.

[12] S. Cook. The complexity of theorem proving procedures. In *Proceedings of Third Annual ACM Symp. on Theory of Computing*, pages 151–158, 1971.

[13] J. M. Crawford and L. D. Auton. Experimental Results on the Crossover Point in Random 3-SAT. *Artificial Intelligence*, 81(1-2):31–57, 1996.

[14] B. Demoen, M. Garcia de la Banda, W. Harvey, K. Marriott and P. Stuckey. An overview of HAL. In *Principles and Practice of Constraint Programming - CP'99, 5th International Conference*, pages 174–188, 1999.

[15] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 415–436, 1996.

[16] J.-F. Dufourd. Algebras and formal specifications in geometric modelling. *The Visual Computer*, 13:131-154, 1997.

[17] J.-F. Dufourd, P. Mathis, and P. Schreck. Formal resolution of geometric constraint systems by assembling. In *Proceedings of the ACM-Siggraph Solid Modelling Conference, Atlanta*, pages 271-284, 1997, ACM Press.

[18] J.-F. Dufourd, P. Mathis, and P. Schreck. Geometric construction by assembling solved subfigures. *Journal of Artificial Intelligence*, 99(1):73-119, Elsevier, 1998.

[19] C. Essert, P. Schreck, and J.-F. Dufourd. Interprétation d'un programme avec multifonctions géométriques. In *Proceedings of 6èmes journées de l'AFIG*, Dunkerque, France, pages 223-232, 1998.

[20] C. Essert, P. Schreck, and J.-F. Dufourd. Sketch-based pruning of a solution space within a formal geometric constraint solver, submitted to *Journal of Artificial Intelligence*, Elsevier.

[21] L.W. Ericson and C.-K. Yap. The design of Linetool, a geometric editor. In *Proceedings of Computational Geometry '88*, LNCS 333, Springer-Verlag, Berlin, pages 83-92, 1988.

[22] J. W. Freeman. *Improvement to propositional satisfiability search*. PhD thesis, Univ. of Philadelphia, 1995.

[23] J.X. Ge, S.C. Chou and X.S. Gao. Geometric constraint satisfaction using optimization methods. *Computer-Aided Design*, 31:867-879, 1999.

[24] H. Gelernter, J.R. Hansen, D.W. Loveland. Empirical explorations of the geometry theorem proving machine. *Computer and Thought*, MacGraw Hill, pages 134-152, 1963.

[25] J.A. Goguen. Modular algebraic specification of some basic geometrical constructions. *Journal of Artificial Intelligence* 37, pages 123-153, 1988.

[26] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[27] D. S. Johnson and M. A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second Dimacs Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[28] G.A. Kramer. A geometric constraint engine. *Artificial Intelligence*, 58:327-360, 1992.

[29] C. Li and Anbulagan. Heuristic Based on Unit Propagation for Satisfiability Problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371, Nagoya, Japan, August 1997.

[30] Light, Lin and Gossard. Variational geometry in CAD. *Computer Graphic*, vol.15 n.3, August 1981.

[31] H. Lamure and D. Michelucci. Solving constraints by homotopy. In *Proceedings of the ACM-Siggraph Solid Modelling Conference*, pages 134-145, 1995, ACM Press.

[32] J. P. Marques Silva. An Overview of Backtrack Search Satisfiability Algorithms. In *Proceedings of the Fifth International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, January 1998. available at `http://rutcor.rutgers.edu/~amai/Proceedings.html`.

[33] P. Mathis. Un système de résolution de contraintes par assemblage en modélisation géométrique, Ph.D. Thesis, Université de Strasbourg, 1997.

[34] B. Mazure, L. Saïs, and É. Grégoire. An efficient technique to ensure the logical consistency of interacting knowledge bases. *International Journal of Cooperative Information Systems*, 6(1):27–36, 1997.

[35] B. Mazure, L. Saïs, and É. Grégoire. Tabu Search for SAT. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 281–285, Rhodes Island, 27-31 July 1997.

[36] B. Mazure, L. Saïs, and É. Grégoire. Boosting Complete Techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22:319–331, 1998.

[37] P. Morris. The Breakout Method for Escaping from Local Minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, pages 40–45, Washington, DC, 18-20 August 1993.

[38] J. Owen. Algebraic solution for geometry from dimensional constraints. In *Proceedings of the 1st ACM Symposium of Solid Modelling and CAD/CAM Applications*, pages 397-407, 1991, ACM Press.

[39] J.M. Scandura, J.H. Durnin, W.H. Wulfeck II. Higher order rule characterization of heuristics for compass and straight edge constructions in geometry. *Artificial Intelligence* 5, pages 149-183, 1974.

[40] H. Schumann and D. Green. Discovering Geometry with a Computer using Cabri Geometre. *ed. Chart-Bratt*, England, 1994.

[41] B. Selman, H. A. Kautz, and B. Cohen. Local Search Strategies for Satifiability Testing. In *Proceedings 1993 DIMACS Workshop on Maximum Clique, Graph Coloring, and Satisfiability*, pages 521–531, 1993.

[42] B. Selman, D. Mitchell, and H. Levesque. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446, San Jose, California, 12-16 July 1992.

[43] G. Sunde. Specification of shape by dimensions and other geometric constraints. In *Proceedings of the Eurographics Workshop on Intelligent CAD systems*, Noordwisjkerout, 1987.

[44] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the IFIP Spring Joint Computer Conference*, pages 329-36, 1963.