A Constraint Programming Approach for Solving Rigid Geometric Systems

Christophe Jermann[1,⋆], Gilles Trombettoni[1],
Bertrand Neveu[2], and Michel Rueher[1]

[1] Université de Nice–Sophia Antipolis, I3S, ESSI
930 route des Colles, B.P. 145, 06903 Sophia Antipolis Cedex, France
{jermann,rueher,trombe}@essi.fr
[2] CERMICS
2004 route des lucioles, 06902 Sophia.Antipolis cedex, B.P. 93, France
neveu@sophia.inria.fr

Abstract. This paper introduces a new rigidification method -using interval constraint programming techniques- to solve geometric constraint systems. Standard rigidification techniques are graph-constructive methods exploiting the degrees of freedom of geometric objects. They work in two steps: a planning phase which identifies rigid clusters, and a solving phase which computes the coordinates of the geometric objects in every cluster. We propose here a new heuristic for the planning algorithm that yields in general small systems of equations. We also show that interval constraint techniques can be used not only to efficiently implement the solving phase, but also generalize former ad-hoc solving techniques. First experimental results show that this approach is more efficient than systems based on equational decomposition techniques.

1 Introduction

Modeling by geometric constraints is a promising method in the CAD field. It allows a user to build a shape by stating geometric constraints in a declarative way. In practice, geometric systems contain numerous rigid subparts. Recursive rigidification techniques [HLS97] allow a bottom-up computation of a rigid system by discovering and aggregating rigid subsystems. We introduce a new rigidification algorithm which is general enough to tackle systems in 2D or 3D. This algorithm yields a decomposition of a rigid system into several subsystems to be solved one by one. The paper aims at showing that this semantic-guided approach proves to be efficient when every subsystem is solved by interval techniques. Moreover, when the subsystems are small, the system may be tractable by symbolic tools.

The paper is organized as follows. The next subsections introduce the problem and recursive rigidification. Section 2 describes the new planning phase we have designed. Section 3 presents a solving phase based on interval techniques. Section 4 provides first experimental results.

⋆ Supported by CNRS and region Provence Alpes Côte d'Azur

R. Dechter (Ed.): CP 2000, LNCS 1894, pp. 233–248, 2000.

1.1 Problem Description

The problem considered in this paper is the computation of all possible positions and orientations of *geometric objects* satisfying *constraints* that make them rigid relative to each other [FH97].

Definition 1 *A* **geometric constraint problem** *is defined by a set of geometric objects and a set of geometric constraints.*

A **geometric object** *is defined by a set of generalized coordinates in a reference system of given dimension such as the Euclidean plane or 3D-space.* **Generalized coordinates** *are parameters defining the position and the orientation of an object.*

A **geometric constraint** *is a relation between geometric objects.*

Examples of geometric objects are points, lines, circles in 2D, or points, lines, spheres, cylinders in 3D space. Geometric constraints can state properties like incidence, tangency, orthogonality, parallelism, distance or angle.

Although the algorithms described in the paper works in 3D space, our implementation is restricted to the following entities in the Euclidean plane:

- Geometric objects: points, lines and circles,
- Geometric constraints: incidence, orthogonality, parallelism, distance and angle.

We will also assume that:

- *The geometric constraints are binary.* This is not a strong limitation since most of geometric constraints are binary. Moreover, no restriction holds on the arity of the corresponding algebraic equations. For instance, a distance constraint involves only two points but the corresponding equation involves four generalized coordinates: the points coordinates.
- *All objects are non-deformable*, that is, the involved generalized coordinates cannot be independent from the reference system. For example, a circle is defined by the two coordinates of the center, but the radius must be constant. Moreover, constraints can only define relations which involve coordinates of objects. For instance, a distance constraint for which the distance parameter is variable cannot be handled.

The second limitation is intrinsic to recursive rigidification which performs rigid-body transformations [JASR99].

1.2 Recursive Rigidification

Recursive rigidification is based on a degree of freedom analysis performed on a weighted geometric constraint graph [HLS97].

Definition 2 *A* **weighted geometric constraint graph** $G = (O, C)$ *is defined as follows:*

- *A vertex $o \in O$ represents a geometric object. Its weight $w(o)$ characterizes the number of its* **degrees of freedom***, i.e., the number of generalized coordinates that must be determined to fix it. For example, a point and a line* [1] *have two degrees of freedom in the Euclidean plane.*
- *An edge $c \in C$ represents a geometric constraint. Its weight $w(c)$ gives the number of parameters that are fixed by the constraint; usually the number of corresponding equations. For example, a distance constraint fixes 1 parameter, and then has weight 1.*

So, the degree of freedom analysis exploits a structural property of the geometric constraint graph, called *structural rigidity*[2].

Definition 3 *Let S be a geometric constraint problem, and let $G = (O, C)$ be the corresponding weighted geometric constraint graph. Let W be the function which computes the difference between the sum of object weights and the sum of constraint weights: $W(G) = \sum_{o \in O}(w(o)) - \sum_{c \in C}(w(c))$.*
In dimension d, the system S is **structurally rigid** *(in short* **s-rigid***) iff:*

- $W(G) = d(d+1)/2$
- *For every sub-graph G' of G, $W(G') \geq d(d+1)/2$*

An s-rigid sub-graph of a geometric constraint graph will be called a **cluster** *in this paper. Intuitively, in 2D, a cluster has 3 degrees of freedom since it can be translated and rotated.*

The structural rigidity is similar to the property P defined in [LM98] and to the *density* notion introduced in [HLS97].

Recursive rigidification creates iteratively a new cluster in the graph: a single node replaces the objects in the created cluster, and arcs connecting several objects in this cluster to one object outside the cluster are condensed into a single arc labeled by the sum of the weights of the synthesized arcs. Figure 1 illustrates the above notions.

The planning phase aims at decomposing the whole system into a sequence of small blocks of equations. It interleaves two steps called *merge* and *extension* steps which produce clusters recursively:

- The **merge step** finds how to form a bigger cluster based on several clusters or geometric objects.
- The **extension step** extends the obtained cluster by adding to it connected objects one by one.

[1] Roughly, the a and b of the corresponding equation: $y = ax + b$

[2] The s-rigidity is a necessary condition to prove rigidity. However, it is not a sufficient condition, except for distance constraints between points in 2D [Hen92]. Hence it should only be considered as a heuristic to detect rigid subparts in a geometric constraint system. Several counterexamples in 2D and 3D show that redundant constraints are the main cause of failure of the guess given by the s-rigidity [LM98].

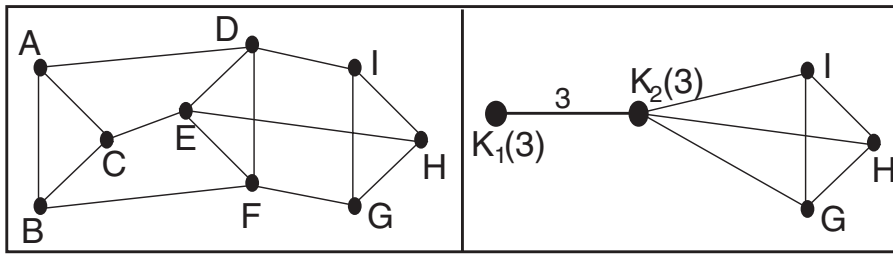

Fig. 1. Graphs associated to a geometric constraint problem.
(Left) All objects are points and distance constraints are posted between points. All edges have weight 1 and all vertices have weight 2 (they are not labeled). Every pair of connected points forms a cluster. Every triangle is also a cluster. Points C, D, E do not form an s-rigid sub-graph since it has 4 degrees of freedom.
(Right) Another graph where triangles (A, B, C) and (D, E, F) have been condensed in two clusters K_1 and K_2. The arc of weight 3 condenses the arcs (A, D), (C, E) and (B, F)

Merge steps would be sufficient to perform a plan. However, a merge step can traverse the whole constraint graph. So, an extension is more or less a heuristic to perform an efficient merge step: one checks incrementally whether the s-rigidity is maintained when only one object is added to the current cluster.

The solving phase, also called construction phase, follows the plan given by the previous phase and computes the coordinates of the geometric objects in every cluster.

1.3 Existing Work

Recursive rigidification techniques have been developed [VSR92,BFH+95,FH93] [FH97,DMS98] to assemble points and lines in 2D systems constrained by distances and angles. [HV95] and [Kra92] describe first attempts to work in 3D. In all these systems, a specific algorithm is used to merge two or three predefined clusters. The possible construction "patterns" appear in a library.

Hoffmann et al. [HLS97,HLS98] have introduced a flow-based algorithm to perform the merge step. This algorithm finds a *minimal dense sub-graph* in a weighted geometric constraint graph, that is, it computes an s-rigid cluster of minimal size (i.e. which has no proper s-rigid sub-graph). Ad-hoc solving methods are used to achieve the actual construction. The algorithm works in any dimension, including 2D and 3D, and can be applied on any type of geometric objects.

The main limitation of Hoffmann's approach comes from the fact that no general method is proposed to perform the solving phase. Symbolic tools could also be considered but are generally not efficient enough to handle these prob-

lems. Of course, ad-hoc solving methods can be used, but they must be defined for every cluster the flow-based algorithm can generate.

1.4 Contribution

In this paper, we propose:

1. A new extension step's heuristic for Hoffmann's planning algorithm [HLS97]. The aim is to generate smaller subsystems of equations.
2. A new and general solving framework which is based on interval techniques. Interval narrowing algorithms [Lho93,HMD97] manage floating intervals and can solve a system of numeric equations by using filtering and domain splitting methods. They are used to compute solutions in every subsystem. Using interval techniques to carry out a construction step of recursive rigidification has two advantages: the approach is general and can replace ad-hoc methods related to specific patterns, and, no solutions are lost.

2 The Planning Phase

The goal of the planning phase is to find an ordering which can solve the constraints in an incremental way. More precisely, the aim is to identify rigid subparts, that can be solved independently (and then assembled).

Hoffmann et al. [HLS97] have introduced an algorithm which achieves such a planning. The main limit of their algorithm comes from the fact that large blocks of constraints have to be added in some cases. Thus the solving process may become very costly. We introduce here a heuristic to limit the number of constraints that are added at each step.

Next sub-section illustrates the principle and the limits of Hoffmann's algorithm on a short example. Afterwards, we detail the proposed heuristic.

2.1 Hoffmann's Algorithm

Hoffmann's planning algorithm builds a reverse tree of clusters called *cluster tree*: the root is the final cluster covering the whole system; the leaves are the geometric objects; there is an arc between a cluster K and all the clusters that have been merged to yield K.

Roughly speaking, the algorithm builds clusters in sequence by interleaving merge and extension steps. It stops as soon as the whole system has been rigidified or when the system cannot be rigidified further. The algorithm updates a geometric weighted graph G_m while achieving merge and extensions steps. For instance, consider an example in 2D made of 15 points and 27 distance constraints between them (see Figure 2 - G_m^0).

The first merge step finds the sub-graph $G =< A, B >$ of G_m. This cluster is extended until a fix-point is reached. The set of adjacent points of this cluster is $\{C, H, I\}$. Since $< A, B, I >$ is s-rigid, I is added to G. The same

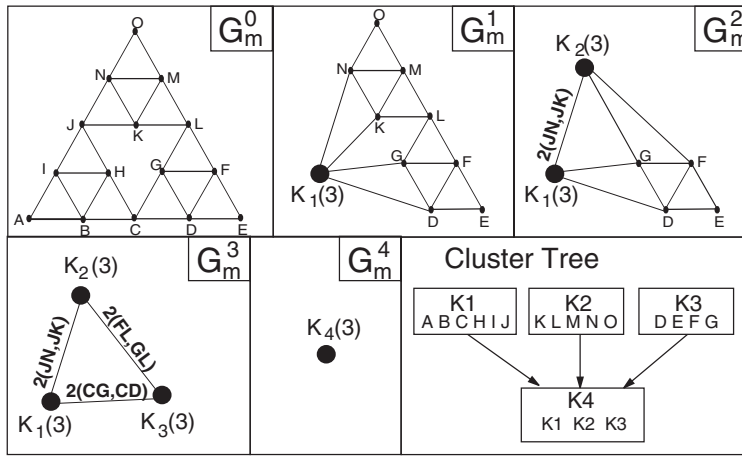

Fig. 2. Snapshots of the graph G_m during the execution of Hoffmann's planning method, and resulting cluster tree

process is performed to add H, C and J to G. Now the set of adjacent points is $\{D, G, N, K\}$. None of these points can be added by extension, so we have reached the fix-point $G =< A, B, I, H, C, J >$. A new vertex K_1 of weight 3 is added and A, B, I, H, C, J are removed from G_m as well as all constraints among them (see Figure 2 - G_m^1). K_1 is placed into the cluster tree and the next merge step is performed. It identifies $< M, N >$ as minimal s-rigid sub-graph of G_m. This cluster is extended to $K_2 =< M, N, O, K, L >$ (see Figure 2 - G_m^2). Note that K_2 could have been extended onto J if $< M, N >$ were identified at the beginning. Finally, G, F, D and E are merged into K_3 (note again that C and L could have been included in K_3) (see Figure 2 - G_m^3).
Since clusters do not share points, inter-cluster constraints (i.e., the distance constraints $dist(J, K)$, $dist(J, N)$, $dist(G, C)$, $dist(G, L)$, $dist(F, L)$ and $dist(C, D)$) are handled by the last merge step (see Figure 2 - G_m^4), for which the solving step can hence be expensive.

Therefore, we propose a new heuristic where clusters can share objects, the aim being to maximize the extension capabilities, and thus, to reduce the number of constraints which have to be handled during the merging steps. This heuristic generalizes previous ad-hoc techniques [BFH+95].

2.2 The Proposed Heuristic

Like Hoffmann's algorithm, Algorithm `Rigidification` interleaves merge and extension steps. To facilitate object sharing it uses the following two graphs:

– The merge graph G_m which corresponds to the graph of clusters used in Hoffmann's algorithm. However, in our algorithm, G_m is used only for the merge step.

– The extension graph G_e which is specially maintained for the extension step. It contains the shared objects as well as the objects which have not yet been included in a cluster.

The algorithm performs three main steps iteratively (within the `while` loop): a *merge step*, an *extension step* and an *update step*.

One merge step is achieved by the `MinimalSRigid`(G_m, d, G_e) function. This function first computes G_1, a minimal dense sub-graph of G_m, with the flow-based algorithm described in [HLS97]. The dense sub-graph is then converted into a sub-graph of G_e and returned. The empty set is returned if G_m contains a single node or if no s-rigid sub-graph can be found.

One extension step extends the cluster G_1 found by the merge step. The `repeat` loop incrementally adds one object to G_1. Objects are added as long as the obtained graph remains s-rigid.

The last step updates the two graphs G_m and G_e.

Updating the Graph G_m

A new cluster K is created in G_m and replaces the included clusters and objects (sub-graph G_2). This is performed by the function `Condense`(G_2, K, G_m) as follows: (a) replace all vertices in G_2, the sub-graph of G_m corresponding to G_1, by a single node K in G_m; (b) combine all arcs from one vertex v of $G_m - G_2$ to vertices of G_2 into one arc from v to K with a weight equal to the sum of the combined arcs.

The newly created cluster K may contain shared variables which have been previously included in other clusters. Coincidence constraints are thus added in G_m to take them into account. Intuitively, coincidence constraints are added to preserve the right number of degrees of freedom in G_m. They state that the different occurrences of a shared object correspond in fact to a single object (function `AddCoincidences`(K, G_m)).

Updating the Graph G_e

The nodes in the newly created cluster K are partitioned into two sets: the *interface objects* that are connected to other objects in G_e and the *internal objects*. Function `RemoveVertices` removes the internal objects since they are s-rigid relative to each other (that results from the fact that they are included in the same cluster K). In the opposite, interface objects remain in G_e since they may potentially be shared by other clusters in further steps.

To maintain the right number of degrees of freedom, the interface objects in G_e must be rigidified. The function `Rigidify` adds *interface constraints* between them in G_e as follows: if the cluster K contains two interface objects o_1 and o_2, a weighted arc (o_1, o_2) is added to make them s-rigid; if there are more than two interface objects, every other object o_i in K is rigidified by adding arcs (o_i, o_1) and (o_i, o_2) (see for coming [JTNR00]).

Algorithm `Rigidification` terminates since the number of objects in G_m decreases at each step. The correction is ensured by the fact that the s-rigidity property is preserved in G_m and G_e as long as the algorithm runs.

Algorithm 1 `Rigidification` (**in** G: Graph; **in** d: Integer; **out** CT: ClusterTree)

```
{G is the initial weighted geometric graph; d is the dimension of the problem (2D, 3D); CT is the plan (cluster tree) that is produced by the algorithm.}
CT ← ∅; G_m ← G; G_e ← G
G_1 ← MinimalSRigid(G_m, d, G_e)   {First merge step}
while G_1 ≠ ∅ do
    {Extension step}
  repeat
    for all o ∈ G_e | ∃o_1 ∈ G_1 and edge (o, o_1) ∈ Edges(G_e) do
      if G_1 ∪ {o} is s-rigid then
          {Add o and corresponding edges to G_1}
        AddVertex(G_1, o, G_e)
      end if
    end for
  until FixPoint {G_1 is no more modified}
  G_2 ← Convert(G_1, G_m)   {G_2 is a sub-graph of G_m corresponding to G_1}
  Condense(G_2, K, G_m)   {Replace G_2 by a new vertex K in G_m}
  AddCoincidences(K, G_m)
  InsertCluster(K, CT)   {Insert K in the cluster tree}
  Rigidify(InterfaceObjects(G_1))   {Add interface constraints of G_1 in G_e}
  RemoveVertices(InternalObjects(G_1), G_e)   {Remove from G_e internal objects and connected arcs}
  G_1 ← MinimalSRigid(G_m, d, G_e)   {Merge step}
end while
```

2.3 Example

Figure 3 illustrates the behavior of Algorithm `Rigidification` on the example introduced in Figure 2.

The first merge and extension steps are similar to Hoffmann's one and yield a sub-graph G_1 containing the points A, B, I, H, C, J. A new vertex K_1 of weight 3 replaces these points in G_m. The internal points A, B, H, I are removed from G_e, along with the internal constraints, but an interface constraint is added between J and C.

The cluster K_2 is then created in the same way. It is condensed into G_m. Since K_2 includes the point J which also belongs to K_1, a coincidence constraint of weight 2 is added in G_m between K_1 and K_2.

The cluster K_3 is finally created. It includes all the remaining points in G_e, and in particular, the interface constraints. Then, the planning phase is finished.

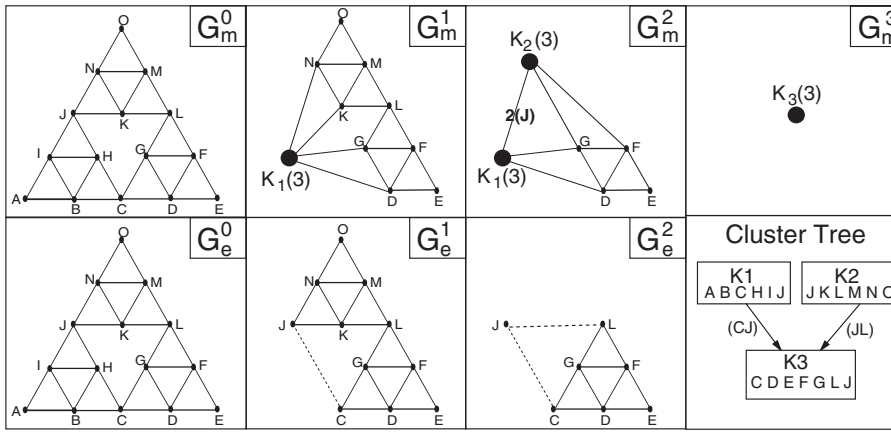

Fig. 3. Snapshots of the graphs G_m and G_e during the run of the algorithm, and obtained cluster tree. Interface constraints are drawn in dotted lines

2.4 Comparing with Hoffmann's Approach

Hoffmann's algorithm uses a single graph to achieve merging steps and extension steps whereas Algorithm `Rigidification` performs the extensions on a specific graph that contains shared objects. Thus, Algorithm `Rigidification` may be able to achieve more extensions. It is important to understand that one extension implies the creation of a system of equations the size of which is not greater than 3 in 2D (6 in 3D), that is, the number of degrees of freedom of the added object. Since this heuristic maximizes the number of extensions steps, it should reduce the solving cost of the merge steps.

For instance, on the previous example, Hoffmann's algorithm builds clusters K_1, K_2 and K_3 before merging them into K_4 (see Figure 2). This corresponds to 9 extension steps and 4 merge steps; the last one will have to merge 3 clusters with 6 distance constraints between them. On the same example, Algorithm `Rigidification` achieves 13 extensions and only 3 merges. None of these steps involve more than 2 distance constraints.

3 The Solving Phase

This section shows how to use interval constraint techniques for solving the tree of clusters built in the planning phase.

Atomic steps of the planning phase generate subsystems of equations, called *blocks* in the paper, that can be solved in sequence. Interval constraint techniques solve every block and yield numeric solutions[3]. When a solution is found

[3] A superset of the solutions is in fact obtained: eliminated parts of the search space never contain any solution, but the remaining non-empty intervals might contain no solution.

in a block, the corresponding variables are replaced by their value in subsequent blocks. When the resolution fails, a backtracking step occurs and another solution is searched for in the previous block. The next subsections detail how to generate the blocks of algebraic equations based on a cluster tree. Different solving processes of the decomposed system are also described.

3.1 Generating the Equations

A directed acyclic graph (DAG) of blocks is created while the cluster tree is built.

- A **block** contains a (sub)system of equations. It includes the equations corresponding to the arcs (geometric constraints) removed from G_e during one merge step or one extension step;
- There is an arc from a block A to a block B if a variable to be instantiated in A also occurs in an equation of B.

Note that interface constraints are considered in the same way as others in this process. Figure 4 provides an example of such a DAG of blocks.

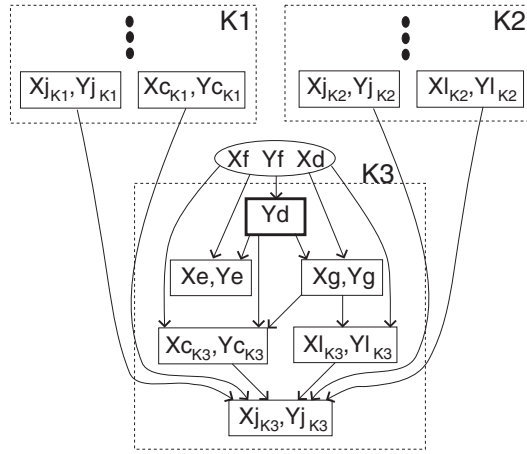

Fig. 4. DAG of blocks associated to the cluster tree of the example in Fig. 3. Blocks are represented by small rectangles showing the computed variables. All the blocks in K_3 are shown. The block at the top of K_3 is created by the merge step (merging D and F). Descendant blocks are created by extension. The last block contains the two interface constraints added during the process

Now, let us detail how interface constraints are handled. Each block computes its own set of variables. The variables corresponding to an interface object are replicated in each cluster where the object occurs. Object J shared by clusters K_1, K_2 and K_3 leads to define variables $x_{J_{K_1}}$, $y_{J_{K_1}}$, $x_{J_{K_2}}$, $y_{J_{K_2}}$, $x_{J_{K_3}}$, $y_{J_{K_3}}$.

Variables $x_{J_{K_1}}$, $y_{J_{K_1}}$ (resp. $x_{J_{K_2}}$, $y_{J_{K_2}}$) are computed when solving clusters K_1 (resp. K_2). When solving cluster K_3, the block computing $x_{J_{K_3}}$, $y_{J_{K_3}}$ is made of the 2 interface constraints $dist(J_3, L_3) = dist(J_2, L_2)$ and $dist(J_3, C_3) = dist(J_1, C_1)$. In this block, the last one in cluster K_3, all the variables except $x_{J_{K_3}}$, $y_{J_{K_3}}$ have already been computed in previous blocks.

Now we will describe the different solving processes based upon interval narrowing techniques.

3.2 Domain Splitting and Filtering per Block

A first approach to solve a DAG of blocks has been described at the beginning of this section. Standard filtering and domain splitting can compute a set of solutions in every block and an inter-block backtracking process is performed when an inconsistent combination of block solutions occurs.

Performing interval narrowing on a block is straightforward, the variables of the entire block being subject to domain splitting and filtering. However, one should pay attention to the inter-block process: the computed values, which will be replaced in a subsequent block, are not floating numbers, but an interval of floating points (even very small, e.g., 10^{-8} large). We could handle such constant intervals by slightly modifying the LNAR function of Numerica [HMD97] [4]. We have chosen another process: the middle point of the reduced constant interval replaces the variable in equations included in subsequent blocks. This middle point heuristic is easy and general. It is correct if the set of intervals obtained at the end is checked, by a filtering process, against all the equations in the entire system. In practice, this final check is very fast since the intervals are really small. Of course, this process does not guarantee to find all solutions but we did never lose any solution in practice on the tested examples.

This approach is very efficient because replacing a variable by a constant simplifies the system of equations.

3.3 Performing Propagation on the Whole System

Another algorithm could be applied that limits domain splitting in one block at a time, but performs filtering by propagation on the whole system. There are two different ways to implement propagation:

1. All the blocks are managed by a standard inter-block backtracking process, just like the pure backtracking algorithm described in the previous subsection. Two systems of equations are thus handled by the interval constraint solver: one system corresponding to the current block to be solved by filtering and domain splitting, and another one which includes the equations in the blocks not yet solved. The second system can be filtered by propagation when an interval is reduced in the first one. This approach will be called *block solving with propagation* in the rest of the paper.

[4] The LNAR function, applied to a variable in an equation, replaces all the other variables by a constant interval and searches for the left most zero.

2. Another approach, where all the system is in a single block, is called *global solving* in the following. It considers the given plan as a heuristic to select the next variables for domain splitting with respect to the decomposition. Their domains are split until the desired precision is reached. Filtering is applied on the full system after each split.

The global solving algorithm is simpler to design than block solving with propagation. However, global solving is less efficient for two reasons. First, all blocks are checked for filtering anyway. Second, it cannot benefit from the middle point heuristic.

Conceptually, it is possible to bring the inter-block backtracking process together with any solving algorithm that can yield several solutions for one block. Symbolic algorithms can be used when no trigonometric equations are required to model the system. Plugging such an algorithm in our inter-block process ensures completeness and could be considered for solving small blocks. On the contrary, classical numerical algorithms should not be used in this decomposition scheme since they provide only one solution per block and those partial solutions may not be combinable.

3.4 Unifying Reference Systems

Once the solving phase is finished, every object in the root cluster has been placed in the final reference system. Only internal objects of clusters have not yet been placed in this system. To do so, rigid-body transformations must be done on the cluster tree. The cluster tree is traversed from the root to the leaves. At each node, one performs a rigid body transformation, in the final reference system, of the coordinates belonging to the internal objects of the cluster. More precisely:

1. The coordinates of the interface objects are known in the final system, as the tree is followed from the root to the leaves, and are used them to compute the transformation coefficients.
2. The internal coordinates of the cluster are then recomputed in the final reference system, based on the obtained transformation matrix.

By traversing the tree in reverse order of its construction, a coordinate of an object is computed only once as an objects becomes internal only once.

4 Experimental Results

This section provides preliminary results on three examples (see Figure 5). Their constraint systems contain points and distance constraints. Since we wanted to compare the time spent for computing all solutions, we have adjusted the distance values in order to obtain a limited number of solutions per problem (128 for Ex1, 64 for Ex2 and 256 for Ex3).

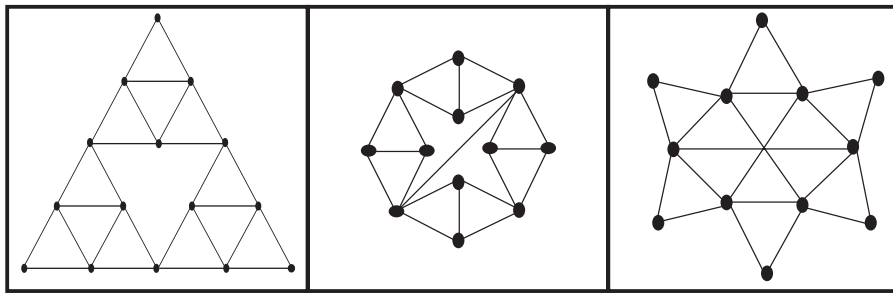

Fig. 5. From left to right, the three 2D examples we consider, made of points and distance constraints: Triangles (Ex1), Diamonds (Ex2) and Hexagon (Ex3)

First, we compare the decomposition obtained by recursive rigidification with a more general equational decomposition which works at the equation and variable level and does not take into account the s-rigidity property of the geometric system [AAJM93,BNT98]. This approach is based upon a structural analysis of the graph of variables and equations, using a maximum matching algorithm and a Dulmage and Mendelsohn decomposition. We also apply our solving techniques on this equational decomposition. For the sake of simplicity, we will use the following abbreviations:

- ED stands for the equational decomposition based on a maximum matching;
- SD1 denotes the decomposition based on shared objets we have introduced in Section 2 (Algorithm `Rigidification`);
- SD2 denotes Hoffmann's rigidification algorithm.

4.1 Maximal Block Size

We can see in Table 1 that the SD1 decomposition leads to smaller blocks than SD2.

The first two examples are decomposed by SD1 into blocks made of 2 equations. With the SD2 decomposition, the maximum block size is 10 or 12. In fact, these blocks of size 10 or 12 can further be decomposed by the equational decomposition technique and the blocks finally solved have a maximum size of 6 in both cases (SD2+ED). For SD1, ED cannot further decompose the obtained blocks. In the third example, a block of size 8 remains in both decompositions and cannot be decomposed by ED anymore.

4.2 Solving with Interval Narrowing Techniques

We provide here the time spent to solve the examples for different decompositions: SD1+ED, SD2+ED, ED and ND. Times for solving with SD1 decomposition are exactly the same as SD1+ED since the plan remains the same with or without applying ED. Times for solving SD2+ED are necessarily better than

Table 1. Size of the biggest equation block obtained by semantic decomposition with shared points (SD1) and without shared points (SD2), the semantic decompositions followed by the equational decomposition (SD1+ED, SD2+ED), equational decomposition on the whole system (ED), and no decomposition (ND, which also represents the size of the complete system)

examples	SD1	SD1+ED	SD2	SD2+ED	ED	ND
Ex1	2	2	10	6	14	26
Ex2	2	2	12	6	10	20
Ex3	8	8	8	8	8	20

times for SD2 alone since SD2+ED provides a better decomposition. For the decomposed systems, we run the 3 methods presented in Sections 3.2 and 3.3: the inter-block backtracking (M1), the inter-block backtracking with propagation (M2), and the global solving which uses the decomposition as a heuristic for choosing the next domain to split (M3).

All experiments were performed on a Pentium III 500, using Ilog Solver [ILO98], with the IlcNumerica library which implements domain filtering by Box-Consistency [BAH94].

Table 2. Results in CPU time (in seconds) for the decompositions SD1+ED, SD2+ED and ED with the solving methods M1, M2 et M3 and for a solving without decomposition (ND)

Examples	**SD1+ED**			**SD2+ED**			**ED**			**ND**
	M1	M2	M3	M1	M2	M3	M1	M2	M3	-
Ex1	17	9	455	43	28	1322	58	29	385	5795
Ex2	1.4	11	77	9	13	178	56	117	467	6640
Ex3	0.9	3.4	289	2.6	12.2	1646	1.5	3.7	533	2744

4.3 Analysis

These results show that:

- Any decomposition is always fruitful: without decomposition, the solving times may be 2 orders of magnitude higher.
- The semantic decomposition (SD1) based on rigidification yields in general smaller blocks than equational decomposition (ED). The performances are better when the maximal block size is smaller.
- Methods M1 and M2 give even better results than M3, which shows the interest of the middle point heuristic.

– The effect of the propagation depends on the problem itself: when many inter-block backtracks occur, like in Ex1, the inter-block constraint propagation (M2) does pay off.

5 Conclusion

This paper has introduced a complete framework for handling geometric constraints. It is composed of:

– A new heuristic for the planning algorithm which allows us to build small subsystems of equations. This semantic-guided phase yields a better decomposition of a rigid system than a syntactic one.
– A solving phase based on interval techniques. This approach is general and does not lose any solution. It is a promising alternative to ad-hoc or classical numeric approaches.

To validate this framework, further experiments have to be performed.

References

AAJM93. Samy Ait-Aoudia, Roland Jegou, and Dominique Michelucci. Reduction of constraint systems. In *Compugraphic*, 1993. 245

BAH94. F. Benhamou, D. Mc Allester, and P. Van Hentenryck. Clp(intervals) revisited. In *Proc. Logic Programming, MIT Press*, 1994. 246

BFH+95. William Bouma, Ioannis Fudos, Christoph Hoffmann, Jiazhen Cai, and Robert Paige. Geometric constraint solver. *Computer Aided Design*, 27(6):487–501, 1995. 236, 238

BNT98. Christian Bliek, Bertrand Neveu, and Gilles Trombettoni. Using graph decomposition for solving continuous csps. In *Principles and Practice of Constraint Programming, CP'98*, volume 1520 of *LNCS*, pages 102–116. Springer, 1998. 245

DMS98. Jean-François Dufourd, Pascal Mathis, and Pascal Schreck. Geometric construction by assembling subfigures. *Artificial Intelligence*, 99:73–119, 1998. 236

FH93. Iaonnis Fudos and Christoph Hoffmann. Correctness proof of a geometric constraint solver. Technical Report TR-CSD-93-076, Purdue University, West Lafayette, Indiana, 1993. 236

FH97. Iaonnis Fudos and Christoph Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics*, 16(2):179–216, 1997. 234, 236

Hen92. Bruce Hendrickson. Conditions for unique realizations. *SIAM J Computing*, 21(1):65–84, 1992. 235

HLS97. Christoph Hoffmann, Andrew Lomonosov, and Meera Sitharam. Finding solvable subsets of constraint graphs. In *Proc. Constraint Programming CP'97*, pages 463–477, 1997. 233, 234, 235, 236, 237, 239

HLS98. Christoph Hoffmann, Andrew Lomonosov, and Meera Sitharam. Geometric constraint decomposition. In B. Brüderlin and D. Roller, editors, *Geometric Constraint Solving and Applications*, pages 170–195. Springer, 1998. 236

HMD97. Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica : A Modeling Language for Global Optimization*. MIT Press, 1997. 237, 243

HV95. C. M. Hoffmann and P. J. Vermeer. A spatial constraint problem. In J.-P. Merlet and B. Ravani, editors, *Computational Kinematics'95*, pages 83–92. Kluwer Academic Publishers, 1995. 236

ILO98. ILOG. Ilog solver reference manual. Technical report, ILOG, 1998. 246

JASR99. R. Joan-Arinyo and A. Soto-Riera. Combining constructive and equational constraint solving techniques. *ACM Transactions on Graphics*, 18(3):35–55, 1999. 234

JTNR00. Christophe Jermann, Gilles Trombettoni, Bertrand Neveu, and Michel Rueher. A constraint programming approach for solving rigid geometric systems. Technical Report 00-43, University of Nice, France, 2000. 239

Kra92. G. Kramer. *Solving Geometric Constraint Systems*. MIT Press, 1992. 236

Lho93. O. Lhomme. Consistency techniques for numeric csps. In *Proc. IJCAI*, Chambery, France, 1993. 237

LM98. Hervé Lamure and Dominique Michelucci. Qualitative study of geometric constraints. In Beat Bruderlin and Dieter Roller, editors, *Geometric Constraint Solving and Applications*, pages 234–258. Springer, 1998. 235

VSR92. A. Verroust, F. Schonek, and D. Roller. Rule oriented method for parametrized computer aided design. *Computer Aided Design*, 24(6):531–540, 1992. 236