Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Programa de Doctorat en Software

PhD Dissertation

Contribution to Geometric Constraint Solving in Cooperative Engineering

Sebastià Vila Marta

Advisor: Robert Joan Arinyo

September 23, 2003

Acknowledgements

I gratefully thank my advisor Prof. Robert Joan-Arinyo for his invaluable guidance, help and advice. I appreciate that he introduced me to geometric constraint solving and that he encouraged me to develop this work.

I would recognize the support of the colleagues from the computer graphics group. To share the weekly discussions with the constraints group have been both enjoyable and helpful. Dr. Vicky Luzón, Dr. Núria Mata, Dr. Núria Pla, Dr. Toni Soto, Josep Suy and Dr. Josep Vilaplana made this possible. Dr. Toni Soto deserves a special acknowledgment for his support and fruitful ideas. To work with him has been a pleasure. I would also thank Laura Fernàndez for her help.

Finally, my warmest gratitude to my family, too large to be enumerated and too important to forget it. Roser and Alba unconditionally supported me during all this time.

Contents

List of Figures

1 Introduction

Geometric models are data structures designed to represent physical properties of objects. The main properties encoded include geometric characteristics and topological properties of physical objects.

Computer Aided Design (CAD) systems are software applications built to help industrial designers during the product design cycle. Because the main activity of CAD systems is related to manage descriptions of objects, geometric models are at the core of such systems.

Due to the central role, geometric models play, they have been extensively studied and different data structures to represent them in the computer have been proposed. Classical references in the field, for example, Requicha [55], Mortenson [49] and Mäntylä [46] discuss geometric modeling schemes, associated operations and their properties.

In the last years a new kind of models have emerged which allow to represent families of objects. These models are known as *parametric models*. The properties exhibited by parametric models resulted in a growing use in modern commercial CAD systems.

A key aspect of parametric models stems from the fact that a natural way of defining parametric objects is based on describing the object by means of a rough sketch. Then the intended object is precisely defined by annotating the sketch with constraints. For this approach to be useful, the problem known as the *geometric constraint solving problem* must be properly solved.

This dissertation focuses on the *constructive geometric constraint solving problem*, an specific instance of the general geometric constraint solving problem, were the solution can be expressed as a set of well-defined geometric operations.

In what follows, we assume that the reader is familiar with the geometric constraint solving field. For generalities and an in-depth review of the literature, the reader is referred to the work of Fudos [19], Soto [58], Durand [15], Solano [57], Jermann [29] and references therein.

1.1 Goals

The specific goals of this work are the following.

1. To define a general architecture for constructive solvers.

   The first goal is to set up a general architecture for constructive solvers.

   Works by Fudos [19], and Soto [58] elaborate on solvers' architectures that naturally arise from the specific constructive geometric constraint solving techniques used. However, the architecture itself never has been investigated.

   In this work we investigate a general model for constructive solvers' architecture that provides a solid foundation to structure software implementations. Moreover, the general architecture defines a framework for the rest of the dissertation.

2. To contribute to the characterization of the domain of solvers based on constructive analysis algorithms.

   Roughly speaking, the domain of a solver is the set of problems that it can effectively solve. Establishing which is the domain of a solving method and which are the properties of this domain are important issues from both a theoretical and a practical point of view. On one hand, since the solving process is computationally expensive, knowing whether a given problem is or is not solvable by an specific solver would be a valuable information. On the other hand, knowing the domain of different solvers would allow to compare their performance.

   We define the tree decomposition of a graph which characterizes a class of geometric constraint graphs. Then we prove that the set of geometric constraint problems solved by three well know analysis algorithms consists precisely in those problems whose graphs are tree decomposable.

3. To develop a theory, based on geometric constraint solving, to solve the problem of model consistency maintenance in CAD systems working with multiple views in concurrent engineering.

   Despite the efforts devoted, the works on this topic reported in the literature only offer incomplete solutions which show several drawbacks, see for example [14], [25] and [26].

   In this thesis we offer a solution based on a set of operations including completion, value transfer and opening a view. Completion is the most difficult operation, it adds new constraints to a solvable under constrained geometric constraint graph to transform it into a well constrained graph, and defines the bridge that relates the second and the third goals.

1.2 Contents

This manuscript is organized as follows. In Chapter 2 we begin by introducing a declarative characterization of a general solver architecture which can be applied to a broad range of constructive geometric constraint solvers. The concepts and architecture defined in this chapter will be used in the rest of the work.

Next, in Chapter 3, we review three already known constructive analysis algorithms: The decomposition and analysis algorithms of Fudos and Hoffman [21], and Owen's decomposition algorithm [50]. We pay special attention to Owen's algorithm and focus on the theoretical foundations of the algorithm, leading to a new formulation. The correctness of Owen's algorithm is also proved under the new formalism.

We discuss the domain characterization problem in Chapter 4. Then we prove that the analysis algorithms studied in Chapter3 have the same domain.

In Chapter 5 we explore the completion problem, that is, how to transform an under constrained constraint graph into a well constrained graph and we offer a solution.

Chapter 6 defines a geometric constraint model suitable to work with multiple views. This new model is built on top of the completion operations defined in Chapter 5.

Finally, Chapter 7 gives the conclusions and offers a list of open problems for future work.

2 Characterization of a Solver Architecture

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

Many techniques have been reported in the literature that provide powerful and efficient methods for solving systems of geometric constraints. For example, see Durand in [15] 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.* The solvers described by Aldefeld [2], Fudos and Hoffmann [4, 18, 19, 21], Soto and Joan-Arinyo [32, 33, 58], Owen [50], Verroust [61, 62], and Brüderlin *et al.* [6–9] are constructive solvers.

According to Fudos *et at.*, constructive solvers have two major components, [21]: the *analyzer* and the *constructor.* The analyzer symbolically determines whether a geometric problem defined by constraints is solvable. If the problem is solvable, the output of the analyzer is a sequence of construction steps, known as the *construction plan*, that places 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.

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 ruler, compass and protractor. In practice, this simple approach solves most useful geometric problems, (see the book of Garling [22]).

Although the constructive geometric constraint solvers proposed in the literature seem to share a common architecture, few efforts have been devoted to characterize it independently of the underlying constraint solving method used.

In this chapter we present a general architecture for constructive geometric constraint solvers. We deliberately avoid focusing on solving methods. First, we identify a set of relevant entities in constructive geometric constraint solving such as abstract geometric constraint problems, abstract construction plans, parameters assignment and index assignment. We present a high level declarative characterization of these entities and their semantics. Next, we identify the three functional units of a geometric constraint solver: the analyzer, the index selector and the constructor. These functional units are specified in terms of the entities that each one manipulates. Lastly, we assemble this functional units in a general architecture for constructive geometric constraint solvers. Most of the work in this chapter has been previously published in [38].

2.1 Preliminaries

In this section we present concepts and notational conventions that will be used throughout all the manuscript.

We assume that a constraint-based design is made of geometric elements like point, lines, circles and arcs of circle. The intended shape is defined by means of constraints like distance between two points, distance from a point to a line, angle between two lines, line-circle tangency and so on.

In what follows, the symbols to represent geometric elements will be taken from the set

$$\mathcal{L}_G = \{p_1, l_1, c_1, p_2, l_2, c_2, \ldots, p_n, l_n, c_n, \ldots\}$$

p_i denoting a point, l_i a straight line and c_i a circle. We assume that the number of different symbols available is unlimited.

Constraints will be represented by predicates relating geometric elements or geometric elements plus a symbolic value called *parameter*. For example,

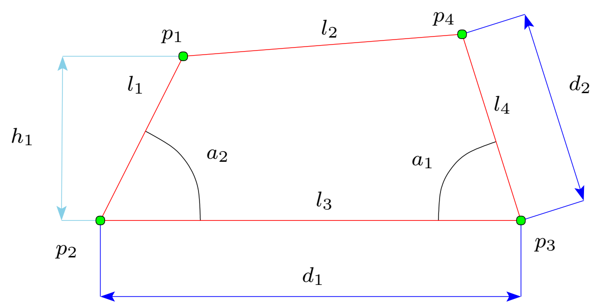
$$\begin{aligned}\mathcal{L}_R = \{&\mathit{onPL}(p, l),\\ &\mathit{distPP}(p_i, p_j, d),\\ &\mathit{distPL}(p_i, l_j, h),\\ &\mathit{angleLL}(l_i, l_j, a), \ldots\}\end{aligned}$$

Predicate names are self explanatory. The predicate $\mathit{onPL}(p, l)$ specifies that point p must lie on line l, $\mathit{distPP}(p_i, p_j, d)$ specifies a point-point distance, $\mathit{distPL}(p_i, l_j, h)$ defines the signed perpendicular distance from a point to a straight line and, $\mathit{angleLL}(l_i, l_j, a)$ denotes the angle between two straight lines. The number and syntax of available constraints are fixed. Symbols d, h and a are parameters. The symbols to represent parameters will be taken from the set

$$\mathcal{L}_P = \{d_1, h_1, a_1, d_2, h_2, a_2, \ldots, d_n, h_n, a_n, \ldots\}$$

d_i denoting a distance between two points, h_i a distance between a point and a line and a_i an angle between two lines. Figure 2.1 shows an example of a constraint-based design and the set of constraints defined between the geometric elements.

This work is centered on constructive geometric constraint solving. Thus, an important entity is the construction plan. To illustrate the concepts, in what follows, we assume that a constructive ruler-and-compass based solver like that reported by Joan-Arinyo and Soto in [32] is available. Therefore,



$onPL(p_1, l_1)$	$onPL(p_1, l_2)$
$onPL(p_2, l_1)$	$onPL(p_2, l_3)$
$onPL(p_3, l_3)$	$onPL(p_3, l_4)$
$onPL(p_4, l_2)$	$onPL(p_4, l_4)$
$distPP(p_2, p_3, d_1)$	$distPP(p_3, p_4, d_2)$
$distPL(p_1, l_3, h_1)$	$angleLL(l_3, l_1, a_2)$
$angleLL(l_3, l_4, a_1)$	

Figure 2.1: Geometric problem defined by constraints.

according to Garling, [22], the basic geometric constructions are

$$\begin{aligned}\mathcal{L}_{CB} = \{&pointXY(x, y),\\ &linePP(p_i, p_f),\\ &lineAP(l, a, p),\\ &circleCR(p, r),\\ &interLL(l_i, lj),\\ &interLC(l, c, s),\\ &interCC(c_i, c_j, s)\}\end{aligned}$$

The meaning of the basic construction names is the usual: point defined by its coordinates, straight line given by an ordered pair of points, straight line through a point at an angle with respect to another line, circle defined by the center and radius, and intersections between straight lines and circles.

The repertoire and syntax of available geometric operations depends on the specific constructive solving approach used and the implementation. However, it is considered fixed. In what follows, we assume that the set of available geometric operations $\mathcal{L}_C$ consists of those given in $\mathcal{L}_{CB}$ plus some

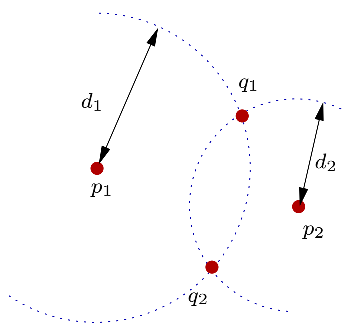


Figure 2.2: Possible placements of a point.

additional simple operations that can be easily expressed as a sequence of operations in $\mathcal{L}_{CB}$, for example *lineLD*(l, h, s), which defines a straight line parallel to another one at a given signed distance, see Joan-Arinyo in [30].

Note that basic intersection operations involving circles may have more that one intersection point. We characterize each intersection point by using an additional *sign parameter*, s, with value in $\{+1, -1\}$. Therefore, this leads to operations like *interLC*(l, c, s) and *interCC*(c_i, c_j, s). For a full definition of the semantics of parameter s see the work of Mata, [47]. The symbols to represent sign parameters will be taken from the set

$$\mathcal{L}_I = \{s_1, s_2, \ldots, s_n, \ldots\}$$

Example 2.1 The intersection between circle $c_1 = \mathit{circleCR}(p_1, d_1)$ and circle $c_2 = \mathit{circleCR}(p_2, d_2)$ in Figure 2.2 are the points $\{q_1, q_2\}$. According to the semantics of sign parameters defined by Mata in [47], we have,

$$\begin{aligned} q_1 &= \mathit{interCC}(c_1, c_2, +1) \\ q_2 &= \mathit{interCC}(c_1, c_2, -1) \end{aligned}$$

$\Diamond$

Given a set of symbols S and a set of values V, a *textual substitution* α is a total mapping from S to V. Let W be a set of predicates and α a textual substitution, we note by $\alpha.W$ the set of predicates obtained by replacing every occurrence of any symbol $s \in S$ occurring in W by $\alpha(s) \in V$.

Example 2.2 Let $S = \{a_1, h_1\}$ be a set of symbols and $V = \mathbb{R}$. Let α be a textual substitution from S to V defined as

$$\alpha(a_1) = 0.57, \quad \alpha(h_1) = 4.0$$

and let W be a set of predicates in $\mathcal{L}_R$ with

$$W = \{onPL(p_1, l_1), angleLL(l_1, l_3, a_1), distPL(p_1, l_3, h_1)\}.$$

Then $\alpha.W$ is

$$\alpha.W = \{onPL(p_1, l_1), angleLL(l_1, l_3, 0.57), distPL(p_1, l_3, 4.0)\}.$$

$\Diamond$

In this Chapter we will also apply textual substitutions to first order logic formulae and other syntactical descriptions.

2.2 Geometric Constraint Problems

We define and declaratively describe two concepts: the abstract geometric constraint problem and the instance of a geometric constraint problem. Abstract entities are exclusively defined in terms of symbols like those in the sets $\mathcal{L}_G$, $\mathcal{L}_P$ and $\mathcal{L}_I$. Instance entities are abstract entities where some of the symbols occurring in them have been replaced by specific values.

2.2.1 The Abstract Problem

An *abstract geometric constraint problem*, or *abstract problem* in short, is a tuple $A = \langle G, C, P\rangle$ where G is a set of symbols in $\mathcal{L}_G$ denoting geometric elements, C is a set of constraints taken from $\mathcal{L}_R$ and defined between elements of G, and P is the set of parameters taken from $\mathcal{L}_P$.

Example 2.3 Consider the sketch with annotated dimension lines shown in Figure 2.1. It can be seen as an abstract problem $A = \langle G, C, P\rangle$ where the set of geometric elements is

$$G = \{p_1, p_2, p_3, p_4, l_1, l_2, l_3, l_4\},$$

C is the set of constraints listed in Figure 2.1 and, the set of parameters is

$$P = \{d_1, d_2, a_1, a_2, h_1\}.$$

$\Diamond$

A convenient way to fully describe an abstract problem is the algorithm-like notation. In this notation, the abstract problem in Example 2.3 can be expressed as

```
gcp A
  param
    $d_1, d_2, a_1, a_2, h_1$ : real
  endparam
  geom
    $p_1, p_2, p_3, p_4$ : point
    $l_1, l_2, l_3, l_4$ : line
  endgeom
  $onPL(p_1, l_1)$
  $onPL(p_1, l_2)$
  $onPL(p_2, l_1)$
  $onPL(p_2, l_3)$
  $onPL(p_3, l_3)$
  $onPL(p_3, l_4)$
  $onPL(p_4, l_4)$
  $onPL(p_4, l_2)$
  $distPP(p_2, p_3, d_1)$
  $distPP(p_3, p_4, d_2)$
  $distPL(p_1, l_3, h_1)$
  $angleLL(l_3, l_1, a_2)$
  $angleLL(l_3, l_4, a_1)$
endgcp
```

Note that an abstract problem defines a family of geometric constraint solving problems parameterized by the set P.

2.2.2 The Instance Problem

A *parameters assignment* is a textual substitution α from a set of parameters P to $\mathbb{R}$.

Let $A = \langle G, C, P \rangle$ be an abstract problem and α be a parameters assignment from P. We say that $\alpha.A = \langle G, \alpha.C, P \rangle$ is an *instance problem* of A. Given an abstract problem, each different parameters assignment defines a different instance problem.

Example 2.4 Consider the abstract problem $A = \langle G, C, P \rangle$ described in the

Example 2.3. An example of parameters assignment α is

$$\begin{aligned}\alpha(a_1) &= -1.222\\ \alpha(a_2) &= 1.0472\\ \alpha(h_1) &= 160.0\\ \alpha(d_1) &= 290.0\\ \alpha(d_2) &= 130.0\end{aligned}$$

A description for the instance problem $\alpha.A$ is

```
**gcp** $\alpha.A$
  **param**
    $d_1, d_2, a_1, a_2, h_1$ : **real**
  **endparam**
  **geom**
    $p_1, p_2, p_3, p_4$ : **point**
    $l_1, l_2, l_3, l_4$ : **line**
  **endgeom**
  $onPL(p_1, l_1)$
  $onPL(p_1, l_2)$
  $onPL(p_2, l_1)$
  $onPL(p_2, l_3)$
  $onPL(p_3, l_3)$
  $onPL(p_3, l_4)$
  $onPL(p_4, l_4)$
  $onPL(p_4, l_2)$
  $distPP(p_2, p_3, 290.0)$
  $distPP(p_3, p_4, 130.0)$
  $distPL(p_2, l_2, 160.0)$
  $angleLL(l_3, l_1, 1.0472)$
  $angleLL(l_3, l_4, -1.222)$
**endgcp**
```

$\Diamond$

Instance problems are no longer parameterized because the parameters have been replaced by the corresponding actual values.

Figure 2.3 shows a graphical representation for the instance problem $\alpha.A$ given in Example 2.4. Now parameters are no longer symbolic but actual values defined by the assignment α. Since Figure 2.3 is a graphical representation of a declarative description of the geometric elements and constraints,

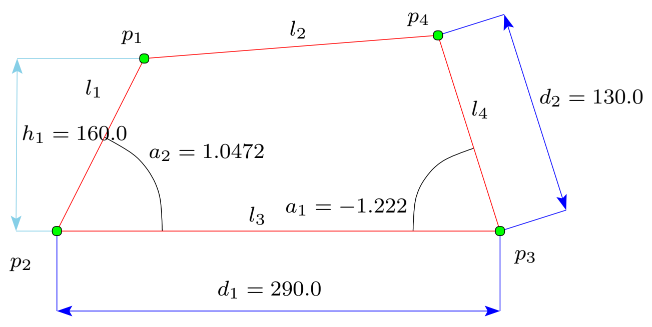


Figure 2.3: Instance problem

the actual geometry is irrelevant. For instance, the actual values of h_1 and d_2 in the figure do not match the values defined by $\alpha(h_1)$ and $\alpha(d_2)$.

Abstract problems precisely describe a set of geometric elements and the constraints that they must fulfill, but abstract problems do not define how to place the geometric elements to satisfy the constraints. In the next section we will present the construction plan which describes how to actually carry out the construction.

2.3 Construction Plan

A construction plan is a procedure that describes how to place the geometric elements with respect to each other. First we formalize the notion of abstract construction plan then we derive the concepts of instance plan and indexed plan.

2.3.1 The Abstract Plan

An *abstract construction plan*, or *abstract plan* in short, is a tuple $S = \langle G, P, L, I \rangle$ where G is a set of symbolic geometric elements taken from $\mathcal{L}_G$, P is a set of parameters taken from $\mathcal{L}_P$, the *index* I is a set of sign parameters taken from $\mathcal{L}_I$, and L is a sequence of basic construction operations taken from $\mathcal{L}_C$ and parameterized by P and I. L defines how to place with respect to each other the elements in G.

Example 2.5

In what follows we refer to the abstract construction plan given below. This plan specifies how to build the geometric object given in Figure 2.1. In the plan, the construction operation *arbitryReferenceSystem*() returns an arbitrary reference system in the plane and the operation *pointXY* returns a point given its coordinates in a reference system.

```
cp S
  param
    $d_1, d_2, a_1, a_2, h_1$ : real
  endparam
  index
    $s_1, s_2$ : sign
  endindex
  geom
    $p_1, p_2, p_3, p_4$ : point
    $l_1, l_2, l_3, l_4$ : line
  endgeom
  $R = arbitraryReferenceSystem()$
  with $R$ do
    $p_2 = pointXY(R, 0, 0)$
    $p_3 = pointXY(R, d_1, 0)$
    $c_1 = circleCR(p_3, d_2)$
    $l_3 = linePP(p_2, p_3)$
    $l_4 = lineAP(l_3, a_1, p_3)$
    $p_4 = interCL(l_4, c_1, s_1)$
    $l_1 = lineAP(l_3, a_2, p_2)$
    $l_8 = lineLD(l_3, h_1, s_2)$
    $p_1 = interLL(l_1, l_8)$
    $l_2 = linePP(p_1, p_4)$
  endwith
endcp
```

Notice that L contains auxiliary symbols, $\{c_1, l_8\}$, which do not belong to G. They are introduced to increase readability. Nonetheless, these symbols can be replaced by their definitions. For instance, symbol l_8 is defined as $l_8 = lineLD(l_3, h_1, s_2)$. If we replace l_8 in the definition of p_1 we have $p_1 = interLL(l_1, lineLD(l_3, h_1, s_2))$. This procedure can be repeated for every auxiliary symbol occurring in L.

Figure 2.4(a) and Figure 2.4(b) illustrate step by step how the plan is interpreted. First an arbitrary point O is chosen to start the construction. This point is labeled p_2, see Figure 2.4(a). Then, the point p_3 whose y coordinate is coincident with O_y

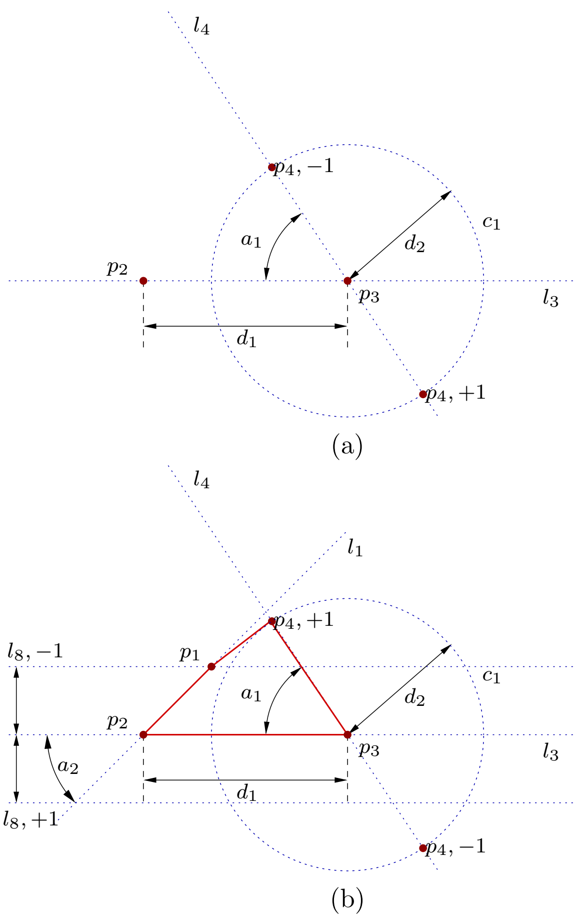


Figure 2.4: Step by step interpretation of the abstract plan given in Example 2.5.

is created at a distance d_2 from p_2. Next the circle c_1, with center on p_3 and radius d_2, the straight line l_3, through points p_2 and p_3, and the line l_4, through point p_3 and at angle a_1 with, l_3 are created. Finally point p_4 is defined as the intersection of c_1 and l_4. Note that there are two possible locations for point p_4. Every possible location is distinguished with the sign s_1, which takes values from the set $\{+1, -1\}$, following the semantics of signs defined by Mata in [47]. Assuming that point p_4 is located in p_4 with $s_1 = -1$ (in the Figure this is noted as $p_4, -1$), and that the line l_8 chosen is l_8 with $s_2 = -1$, Figure 2.4(b) shows the interpretation of the rest of the plan. $\Diamond$

An abstract construction plan is parameterized by two sets: P and I.

In the following sections we present the concepts of instance plan and indexed plan. In an instance plan we fix the values of parameters in P and in an indexed plan we fix the values of signs in I.

2.3.2 The Instance Plan

An abstract plan can be instantiated by applying a parameters assignment in the same way it has been done for abstract problems. Let $S = \langle G, P, L, I \rangle$ be an abstract plan and α a parameters assignment for P. The instance plan $\alpha.S$ is defined as $\alpha.S = \langle G, P, \alpha.L, I \rangle$.

Example 2.6 Applying the parameters assignment given in Example 2.4 to the abstract plan in Example 2.5, yields the instance plan

```
cp $\alpha.S$
  param
    $d_1, d_2, a_1, a_2, h_1$ : real
  endparam
  index
    $s_1, s_2$ : sign
  endindex
  geom
    $p_1, p_2, p_3, p_4$ : point
    $l_1, l_2, l_3, l_4$ : line
  endgeom
  $R = \mathit{arbitraryReferenceSystem}()$
  with $R$ do
    $p_2 = \mathit{pointXY}(R, 0, 0)$
    $p_3 = \mathit{pointXY}(R, 290.0, 0)$
    $c_1 = \mathit{circleCR}(p_3, 130.0)$
    $l_3 = \mathit{linePP}(p_2, p_3)$
    $l_4 = \mathit{lineAP}(l_3, -1.222, p_3)$
    $p_4 = \mathit{interCL}(l_4, c_1, s_1)$
    $l_1 = \mathit{lineAP}(l_3, 1.0472, p_2)$
    $l_8 = \mathit{lineLD}(l_3, 160.0, s_2)$
    $p_1 = \mathit{interLL}(l_1, l_8)$
    $l_2 = \mathit{linePP}(p_1, p_4)$
  endwith
endcp
```

Figure 2.5 shows the four possible evaluations of the instance plan in Example 2.6 obtained by changing the values of signs s_1 and s_2. $\Diamond$

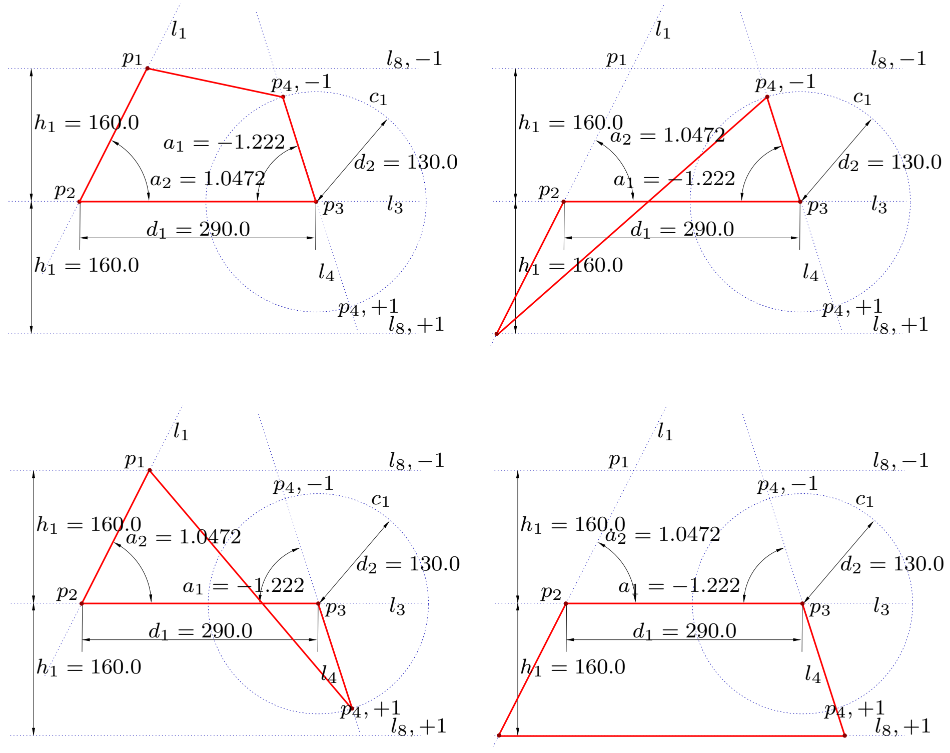


Figure 2.5: The four possible evaluations of the instance plan in Example 2.6

2.3.3 Indexed Plan

An *index assignment*, denoted ι, is a textual substitution from an index I to the set $\{+1, -1\}$.

Let $S = \langle G, P, L, I \rangle$ be an abstract plan and ι an index assignment from I. The *indexed plan* $\iota.S$ is defined as $\iota.S = \langle G, P, \iota.L, I \rangle$.

Example 2.7 Let the index assignment ι be

$$\iota(s_1) = -1, \quad \iota(s_2) = +1.$$

Applying ι to the abstract plan in Example 2.5, yields the indexed plan

cp S

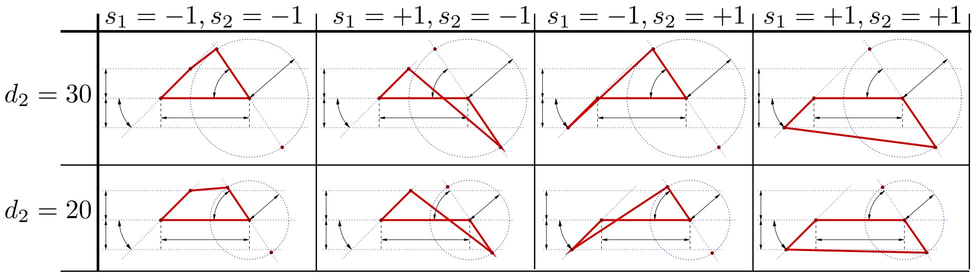


Figure 2.6: Distinct constructions encoded into the same abstract plan.

```
  param
      $d_1, d_2, a_1, a_2, h_1$ : real
  endparam
  index
      $s_1, s_2$ : sign
  endindex
  geom
      $p_1, p_2, p_3, p_4$ : point
      $l_1, l_2, l_3, l_4$ : line
  endgeom
  $R = arbitraryReferenceSystem()$
  with $R$ do
      $p_2 = pointXY(R, 0, 0)$
      $p_3 = pointXY(R, d_1, 0)$
      $c_1 = circleCR(p_3, d_2)$
      $l_3 = linePP(p_2, p_3)$
      $l_4 = lineAP(l_3, a_1, p_3)$
      $p_4 = interCL(l_4, c_1, -1)$
      $l_1 = lineAP(l_3, a_2, p_2)$
      $l_8 = lineLD(l_3, h_1, +1)$
      $p_1 = interLL(l_1, l_8)$
      $l_2 = linePP(p_1, p_4)$
  endwith
endcp
```

Figure 2.6 shows two different families of objects generated by changing the value of parameter d_2 in the plan. $\Diamond$

Note that the application of a parameters assignment α and an index assignment ι to an abstract plan S commute. That is $\alpha.\iota.S = \iota.\alpha.S$.

2.4 Characteristic Formulae

We will represent geometric constraint problems and construction plans by means of first order logic formulae. This will allow us to precisely characterize the set of placements of the geometric elements for which the set of constraints hold and, the set of placements actually generated by a construction plan.

2.4.1 Geometric Problems

Let $A = \langle G, C, P\rangle$ be an abstract geometric constraint problem with

$$C = \{c_1, c_2, \ldots, c_m\}$$

Then the *characteristic formula* of A is the first order logic formula,

$$\Psi(A) \equiv \bigwedge_{i=1}^{m} c_i$$

where the geometric elements of G and the parameters of P occurring in Ψ are interpreted as free variables.

Example 2.8 The characteristic formula of the abstract problem A given in the Example 2.3 is

$$\begin{aligned}\Psi(A) \equiv\ & (\mathit{onPL}(p_1, l_1) \wedge \mathit{onPL}(p_1, l_3) \wedge \\ & \mathit{onPL}(p_2, l_1) \wedge \mathit{onPL}(p_2, l_4) \wedge \\ & \mathit{onPL}(p_3, l_3) \wedge \mathit{onPL}(p_3, l_4) \wedge \\ & \mathit{onPL}(p_4, l_2) \wedge \mathit{onPL}(p_4, l_4) \wedge \\ & \mathit{distPP}(p_2, p_3, d_1) \wedge \mathit{distPP}(p_3, p_4, d_2) \wedge \\ & \mathit{distPL}(p_1, l_3, h_1) \wedge \mathit{angleLL}(l_1, l_3, a_2) \wedge \\ & \mathit{angleLL}(l_4, l_3, a_1))\end{aligned}$$

$\Diamond$

Let α be a parameters assignment for P, and $\alpha.A$ the corresponding instance problem. Then the first order formula $\Psi(\alpha.A)$ expresses the instance problem. Note that a textual substitution α can be applied to both an abstract problem or to a first order logic formula. Therefore the relation $\Psi(\alpha.A) = \alpha.\Psi(A)$ is well defined.

Example 2.9 The characteristic formula of the instance problem in Example 2.4 is

$$\begin{array}{rcl}\Psi(\alpha.A) & \equiv & (\mathit{onPL}(p_1, l_1) \wedge \mathit{onPL}(p_1, l_3) \wedge \\ & & \mathit{onPL}(p_2, l_1) \wedge \mathit{onPL}(p_2, l_4) \wedge \\ & & \mathit{onPL}(p_3, l_3) \wedge \mathit{onPL}(p_3, l_4) \wedge \\ & & \mathit{onPL}(p_4, l_2) \wedge \mathit{onPL}(p_4, l_4) \wedge \\ & & \mathit{distPP}(p_2, p_3, 290.0) \wedge \\ & & \mathit{distPP}(p_3, p_4, 130.0) \wedge \\ & & \mathit{distPL}(p_1, l_3, 160.0) \wedge \\ & & \mathit{angleLL}(l_3, l_1, 1.0472) \wedge \\ & & \mathit{angleLL}(l_3, l_4, -1.222))\end{array}$$

$\Diamond$

A *geometry assignment* or *anchor* κ is a textual substitution such that assigns an actual geometry to each geometric element in a set of geometry symbols G.

Let $A = \langle G, C, P\rangle$ be an abstract problem and κ an anchor for G. We define $\kappa.A$ as $\langle G, \kappa.C, P\rangle$.

Example 2.10 If we represent a point by the pair $(x, y) \in \mathbb{R}^2$ and a straight line by (a, b, c), the coefficients of the normal form $ax + by + c = 0$ with $a^2 + b^2 = 1$, then an example of anchor κ is

$$\begin{array}{rcl}\kappa(p_1) & = & (92.38, 160) \\ \kappa(p_2) & = & (0, 0) \\ \kappa(p_3) & = & (290, 0) \\ \kappa(p_4) & = & (245.54, 122.16) \\ \kappa(l_1) & = & (-0.87, 0.5, 0) \\ \kappa(l_2) & = & (-0.24, -0.97, 177.48) \\ \kappa(l_3) & = & (0, -1, 0) \\ \kappa(l_4) & = & (0.94, 0.34, -272.51)\end{array}$$

The characteristic formula Ψ after applying the anchor κ to the instance problem

$\alpha.A$ in Example 2.9 is

$$\begin{aligned}
&\Psi(\kappa.\alpha.A)\\
&\quad\equiv\\
&(\mathit{onPL}((92.38, 160), (-0.87, 0.5, 0)) \wedge\\
&\mathit{onPL}((92.38, 160), (0, -1, 0)) \wedge\\
&\mathit{onPL}((0, 0), (-0.87, 0.5, 0)) \wedge\\
&\mathit{onPL}((0, 0), (0.94, 0.34, -272.51)) \wedge\\
&\mathit{onPL}((290, 0), (0, -1, 0)) \wedge\\
&\mathit{onPL}((290, 0), (0.94, 0.34, -272.51)) \wedge\\
&\mathit{onPL}((245.54, 122.16),\\
&\qquad\quad (-0.24, -0.97, 177.48)) \wedge\\
&\mathit{onPL}((245.54, 122.16),\\
&\qquad\quad (0.94, 0.34, -272.51)) \wedge\\
&\mathit{distPP}((0, 0), (290, 0), 290.0) \wedge\\
&\mathit{distPP}((290, 0), (245.54, 122.16), 130.0) \wedge\\
&\mathit{distPL}((92.38, 160), (0, -1, 0), 160.0) \wedge\\
&\mathit{angleLL}((0, -1, 0),\\
&\qquad\qquad (-0.87, 0.5, 0), 1.0472) \wedge\\
&\mathit{angleLL}((0, -1, 0),\\
&\qquad\qquad (0.94, 0.34, -272.51), -1.222))
\end{aligned}$$

$\diamondsuit$

Notice that α and κ commute, that is, $\kappa.\alpha.A = \alpha.\kappa.A$. Moreover, we say that two distinct anchors κ_1 and κ_2 are equivalent ($\kappa_1 \equiv_r \kappa_2$) iff κ_1 can be obtained from κ_2 by applying two rigid transformations: one rotation and one translation.

Let κ be an anchor for G. The quotient set of anchors for which the formula $\Psi(\kappa.\alpha.A)$ holds modulo the equivalence relation $\equiv_r$

$$V(\alpha.A) = \{\kappa \quad | \quad \Psi(\kappa.\alpha.A)\}/\equiv_r$$

define the set of anchors which are solution to the instance geometric constraint problem $\alpha.A$. We refer to the anchors in $V(\alpha.A)$ as *realizations* of the instance problem $\alpha.A$.

Figure 2.5 shows a graphical representation of the set of realizations $V(\alpha.A)$ for the instance problem $\alpha.A$ in Example 2.4.

2.4.2 Construction Plans

Let $S = \langle G, P, L, I \rangle$ be an abstract construction plan with $L = \{o_1, o_2, \ldots, o_n\}$. The *characteristic formula* of S is the first order logic formula,

$$\Phi(S) \equiv \bigwedge_{i=1}^{n} o_i$$

where the geometric elements of G, the parameters of P and signs of I occurring in Φ are considered free variables.

Example 2.11 The characteristic formula of the abstract plan S given in Example 2.5 is

$$\begin{array}{rcl}
\Phi(S) & \equiv & (p_2 = \mathit{pointXY}(R, 0, 0) \\
& \wedge & p_3 = \mathit{pointXY}(R, d_1, 0) \\
& \wedge & c_1 = \mathit{circleCR}(p_3, d_2) \\
& \wedge & l_3 = \mathit{linePP}(p_2, p_3) \\
& \wedge & l_4 = \mathit{lineAP}(l_3, a_1, p_3) \\
& \wedge & p_4 = \mathit{interCL}(l_4, c_1, s_1) \\
& \wedge & l_1 = \mathit{lineAP}(l_3, a_2, p_2) \\
& \wedge & l_8 = \mathit{lineLD}(l_3, h_1, s_2) \\
& \wedge & p_1 = \mathit{interLL}(l_1, l_8) \\
& \wedge & l_2 = \mathit{linePP}(p_1, p_4))
\end{array}$$

$\Diamond$

Let α be a parameters assignment for P, and $\alpha.S$ the corresponding instance plan. Then the first order formula $\Phi(\alpha.S)$ expresses the instance plan. Note that $\Phi(\alpha.S) = \alpha.\Phi(S)$ trivially holds.

Example 2.12 The characteristic formula of the instance plan in Example 2.6

is

$$\begin{aligned}\Phi(\alpha.S) \;\equiv\; & (p_2 = \mathit{pointXY}(R,0,0)\\ \wedge\; & p_3 = \mathit{pointXY}(R,290.0,0)\\ \wedge\; & c_1 = \mathit{circleCR}(p_3,130.0)\\ \wedge\; & l_3 = \mathit{linePP}(p_2,p_3)\\ \wedge\; & l_4 = \mathit{lineAP}(l_3,-1.222,p_3)\\ \wedge\; & p_4 = \mathit{interCL}(l_4,c_1,s_1)\\ \wedge\; & l_1 = \mathit{lineAP}(l_3,1.0472,p_2)\\ \wedge\; & l_8 = \mathit{lineLD}(l_3,160.0,s_2)\\ \wedge\; & p_1 = \mathit{interLL}(l_1,l_8)\\ \wedge\; & l_2 = \mathit{linePP}(p_1,p_4))\end{aligned}$$

$\Diamond$

Let $S = \langle G, P, L, I\rangle$ be an abstract plan and κ an anchor for G. We define $\kappa.S$ as $\langle G, P, \kappa.L, I\rangle$.

Example 2.13 Let κ be the anchor in Example 2.10 and $\alpha.S$ the instance plan in Example 2.6. The characteristic formula Φ after applying the anchor κ to the instance problem $\alpha.S$ is

$$\begin{aligned}&\Phi(\kappa.\alpha.S)\\ &\quad\equiv\\ &((0,0) = \mathit{pointXY}(R,0,0)\wedge\\ &(290,0) = \mathit{pointXY}(R,290.0,0)\wedge\\ &c_1 = \mathit{circleCR}((290,0),130.0)\wedge\\ &(0,-1,0) = \mathit{linePP}((0,0),(290,0))\wedge\\ &(0.94,0.34,-272.51) =\\ &\quad \mathit{lineAP}((0,-1,0),-1.222,(290,0))\wedge\\ &(245.54,122.16) =\\ &\quad \mathit{interCL}((0.94,0.34,-272.51),c_1,s_1)\wedge\\ &(-0.87,0.5,0) =\\ &\quad \mathit{lineAP}((0,-1,0),1.0472,(0,0))\wedge\\ &l_8 = \mathit{lineLD}((0,-1,0),160.0,s_2)\wedge\\ &(92.38,160) = \mathit{interLL}((-0.87,0.5,0),l_8)\wedge\\ &(-0.24,-0.97,177.48) =\\ &\quad \mathit{linePP}((92.38,160),(245.54,122.16)))\end{aligned}$$

$\Diamond$

Let κ be an anchor for G and α a parameters assignment for P. The quotient set of anchors for which there is an index assignment ι such that the formula

$\Phi(\iota.\kappa.\alpha.S)$ holds modulo equivalence relation $\equiv_r$

$$V(\alpha.S) = \{\kappa \quad | \quad \exists\, \iota\ \Phi(\iota.\kappa.\alpha.S)\} / \equiv_r$$

define the set of anchors which are computed by the instance plan $\alpha.S$. We refer to the anchors in $V(\alpha.S)$ as *indexed anchors* of the instance plan $\alpha.S$.

Figure 2.5 shows a graphical representation of the set of indexed anchors $V(\alpha.S)$ for the instance plan $\alpha.S$ in Example 2.6.

Given an index assignment ι and a parameters assignment α, there is at most one anchor κ for which $\Phi(\kappa.\iota.\alpha.S)$ holds.

2.5 Constructive Solvers

In the preceding sections we have identified a set of entities relevant in the constructive geometric constraint solving process: abstract problems, parameters assignments, instance problems, abstract plans, instance problems, index assignments and anchors. In this section we present an architecture for constructive geometric constraint solvers based on three functional units: the analyzer, the index selector and the constructor. We will specify the functionality of each unit by stating the input, the output and the relationships between them.

2.5.1 The Analyzer

The *analyzer* is the functional unit that computes an abstract plan $S = \langle G, P, L, I \rangle$ from an abstract problem $A = \langle G, C, P \rangle$. The relationship between the abstract problem A and the abstract plan S established by the definition of the analyzer is that the sets G and P are the same in A and S.

There are three important concepts related to analyzers that should be defined.

Definition 2.1 *The* analyzer domain *is the set of abstract problems A for which an analyzer computes a construction plan S.*

The domain of a given analyzer algorithm is important because it is a measure of the solver power. We will characterize the domain of some constructive analyzers in Chapter 4.

Definition 2.2 *We say that an analyzer is* correct *if and only if for every abstract problem A in its domain, the analyzer computes an abstract plan S, and for every parameters assignment α, the relationship $V(\alpha.S) \subseteq V(\alpha.A)$ holds. That is, each anchor for which the construction plan S is feasible corresponds to one realization of the instance problem A.*

Correctness of a solver is a must. Obviously we are not interested in analyzers which are not correct. We assume that analysis algorithms found in the literature are correct.

Notice that this notion of correctness is different from that introduced by Fudos in [19] and also applied by Soto in [58].

Definition 2.3 *We say that an analyzer is* ideal *if and only if for every abstract problem A in its domain, and for every parameters assignment α, the set of anchors computed by the construction plan S and the set of realizations of the instance problems A are coincident, that is $V(\alpha.S) = V(\alpha.A)$.*

An ideal analyzer is always correct. Although ideality is a desirable property, an analyzer can still be useful even if it is not ideal. In Chapter 6 we rely on this property to define important operations to synchronize multiple views. The analyzer described in [32] is ideal.

Example 2.14 Since the abstract plan S in Example 2.5 has been generated from the abstract problem A in Example 2.3 by an ideal analyzer, the set of indexed anchors of the instance plan $\alpha.S$ in Example 2.6 and the set of realizations of the instance problem $\alpha.S$ in Example 2.4 are the same set. Figure 2.5 shows this set. $\diamondsuit$

2.5.2 The Index Selector

An *index selector* is a functional unit characterized by its output which is an index assignment ι. An index assigment is always associated to a given abstract plan. Therefore the plan must also be considered an input to the index selector. Moreover, additional input data must be considered depending on the selection method that the functional unit implements. Here we enumerate some methods.

1. A *trivial* index selector returns an index assignment ι fixed *a priori.* For instance, $\iota(s) = +1$ for all s in I.

2. An index assignment ι from $I = \{s_1, \dots, s_n\}$ can be represented by the binary value $d_1 d_2 \dots d_n$ where $d_i = 0$ if $\iota(s_i) = -1$ and $d_i = 1$ if $\iota(s_i) = 1$. The order relation in binary numbers induces an order in the index assignments. Therefore, we can define a *successor* (*predecessor*) index selector to compute the next (previous) index assignment ι' from a given index assignment ι.

3. An *anchor-based* index selector computes an index assignment ι from an anchor κ and an abstract plan $S = \langle G, P, L, I \rangle$. The output is an index such that defines a realization where the placement of geometric elements preserves the relative orientations defined by the anchor κ. This kind of selector has been studied by several authors in [4, 16, 44].

See the work of Luzón in [44] for an extensive analysis of methods for implementing index selectors.

2.5.3 The Constructor

The *constructor* is the functional unit that computes an anchor κ from an abstract plan S, a parameters assignment α and an index assignment ι. The anchor κ is a realization in $V(\alpha.A)$ provided that the abstract plan S has been computed from the abstract problem A by a correct analyzer.

2.6 Solvers Software Architecture

In this section we present a software architecture useful for building a geometric constraint solving tool-box. The aim of such a tool-box is to provide the software engineer with a set of tools to design and implement software applications based on constraint solving.

The architecture has functional units and data entities. The data entities are geometric constraint problems, constructions plans, parameters assignments, geometry assignments and index assignments. The functional units are analyzers, index selectors and constructors. All these components relate each other following the data-flow diagram shown in Figure 2.7.

This architecture exhibits a number of advantages:

1. The architecture is precisely and concisely defined.

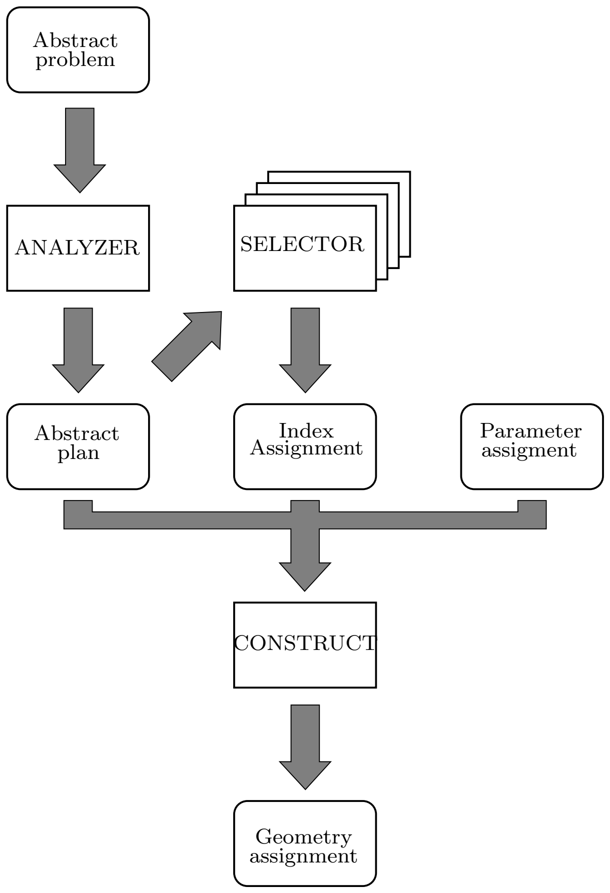


Figure 2.7: Architecture data-flow diagram.

2. It is independent of any particular implementation of the functional units. All what is needed is to define the specific grammar and semantics of $\mathcal{L}_R$ and $\mathcal{L}_C$ given in Section 2.1.

3. It is well suited for interactive applications. For instance, when the user changes the parameters assignment, the analysis and index selection can be skipped and only the construction step have to be carried out. In general, for a given change in any data entity, the data-flow diagram in Figure 2.7 defines which entities must be recomputed.

4. The functional units are reusable to solve problems which are not geometric constraint solving problems but are related. For example, in [31] the tool-box is applied to deal with requirements of concurrent

engineering applications.

Solvers reported in [2, 4, 9, 32, 50] can be implemented following the proposed architecture.

We have identified a set of basic geometric operations, $\mathcal{L}_C$, which allows to express in a unified way the construction plans generated by the solvers given in [4, 32, 50]. The corresponding analyzers have been federated in the architecture, sharing index selectors and constructors.

2.7 Geometric Models

On the one hand geometric models as usually defined, see for instance Requicha in [55], are intended to represent a unique solid. On the other hand, constraint based models represent a family of models. To make constraint based models a useful tool in conventional solid modeling they should be restricted to represent a unique object. In what follows, we define the concept of geometric model from this point of view. In the definition we use the architectural elements of geometric constraint solvers defined in this Chapter.

Definition 2.4 *Given a particular analyzer, a* geometric model *M is a tuple $M = \langle A, \alpha, \iota \rangle$ where A is an abstract system, α is a parameters assignment for A, and ι is an index assignment relative to the abstract plan generated by the analyzer.*

To define the geometric model it is necessary to fix the analyzer because the index meaning depends on the particular abstract plan generated. Given the four elements of a geometric model as defined before a unique object can be computed. Consider $M = \langle A, \alpha, \iota \rangle$ a geometric model. Then by applying an analyzer to A, compute the abstract plan S. Now compute the anchor $\alpha.\iota.S$. This anchor is the object represented by the model.

Since a model includes an index assignment which is analyzer-dependant, transferring a model between differents analyzers is difficult. A way to deal with this problem would include in the model some information encoding the actual index assignment in an analyzer-independant way.

2.8 Summary

We have presented a general architecture for constructive geometric constraint solvers. The architecture is based on three functional units: the analyzer, the index selector and the constructor. Functional units have been precisely defined in terms of their input and output. Every functional unit have a particular type of input and output data. All these data entities have been characterized: the abstract problem, the abstract construction plan, the parameters assignment and the index assignment.

Software implementations can be developed by applying the proposed architecture straightforward. We have enumerated a number of benefits obtained by using this architecture when developing constructive geometric constraint solvers.

Based on this architecture we have defined the concept of solid model. This concept adapts the classical definition of a solid model to the particular characteristics of constructive geometric constraint models.

To illustrate the concepts, we have used functional capabilities which are specific to the ruler-and-compass constructive geometric constraint solving technique. However, the concepts apply to any constructive approach. All what is needed is to replace the set of geometric elements, the set of constraints available and the set of basic constructions with those in the constructive approach of interest.

3 Analysis Methods

As defined in Chapter 2, an analysis method is an algorithm that computes an abstract plan from an abstract geometric constraint problem. In this chapter we study three main constructive analysis methods. The three analysis methods reviewed here are *reduction* and *decomposition* of Fudos and Hoffmann, [19–21], and also the decomposition method due to Owen, [50].

To describe Fudos and Hoffmann reduction and decomposition methods, we introduce the concept of *set decomposition of a graph*. This concept allows us to study both methods from an unified point of view.

Owen's decomposition method is completely reformulated. We define the *deficit* function and prove some properties. Using these results we prove the correctness of the reformulated Owen's method. This new version of Owen's algorithm plays an important role in Chapter 4 where the domains of analysis methods are studied.

Most of the work of this chapter has been previously published in references [35, 37, 39].

3.1 Abstract Problems and Graphs

Abstract problems are naturally represented as graphs. Many algorithms require this kind of representation as input data. In this section we first introduce some concepts related to graphs and later we describe how an abstract problem is represented as a graph. Then we explore when a geometric constraint graph is well constrained and, finally, we introduce the concept of set decomposition.

3.1.1 Graph Concepts

First we recall some basic terminology of graph theory that will be used in the rest of the chapter. For an extensive treatment see the books by Chartrand and Lesniak [10], Gross and Yellen [24], Even [17], Leeuwen [60], and the report of Hopcroft and Tarjan [28].

A *graph* is a structure which consists of a set of *vertices (or nodes)* $V = \{v_1, v_2, \dots\}$ and a set of *edges (or bonds)* $E = \{e_1, e_2, \dots\}$; each edge in E is *incident* to the elements of an unordered pair of vertices (u, v) which are necessarily distinct. Both V and E are assumed to be finite. As usual, we write $E(G)$ (and $V(G)$) to denote the set of edges (and vertices) of the graph G. If $e \in E(G)$ is an edge of G, then $V(e)$ is the set of vertices to which e is incident.

We say that a graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$. Let $G = (V, E)$ a graph and let $V_i \subseteq V$ a subset of vertices. Then the subgraph of G induced by V_i is the graph $G_i = (V_i, E_i)$ where $E_i = \{e | V(e) \subseteq V_i\}$.

Given two graphs $G = (V, E)$ and $G' = (V', E')$ we define the *union* of both as the graph $G \cup G' = (V \cup V', E \cup E')$.

A *path* is a sequence of edges $e_1, e_2, \dots, e_n$ such that:

1. e_i and e_{i+1} have a common endpoint.
2. if e_i is not the first or last edge then it shares one of its endpoints with e_{i-1} and another with e_{i+1}.

A graph $G = (V, E)$ is said to be *connected* if every vertex is connected to every other vertex by at least one path of edges. We say that a vertex a of a connected graph G is an *separation vertex or node* (*articulation vertex or node*) if by removing a, the graph splits into two or more disconnected subgraphs. If a is an articulation node in G, then there are two vertices u and v different from a such that a is on every path connecting u and v. Let a be a separation vertex of a connected graph $G = (V, E)$, then E can be divided into the *separation classes* induced by a, say $E_1, E_2, \dots, E_n$, defined as follows. Two edges are in the same separation class E_i iff there is a path using both edges and not containing a except, possibly, as an endpoint.

A graph with no articulation vertices is called *biconnected.* If u and v are arbitrary non adjacent vertices of a biconnected graph G, then there are at least two vertex disjoint paths in G connecting them. A connected graph can be uniquely decomposed into biconnected components by splitting it at separation vertices. Aho *et al.*, [1], reported a depth first algorithm that efficiently computes such a decomposition.

Let a and b be two vertices in a biconnected graph G. The edges of G can be divided into the *separation classes* induced by $\{a, b\}$, say $E_1, E_2, \ldots, E_n$, defined as follows, [28]. Two edges are in the same separation class E_i if there is a path using both edges and not containing a or b except, possibly, as endpoints. If the two vertices a and b divide the edges into more than two separation classes, then the pair $\{a, b\}$ is a *separation pair* (*articulation pair*) of G. Moreover, if $\{a, b\}$ divides the edges into two separation classes, each containing more than one edge, then $\{a, b\}$ is also a separation pair.

A *triconnected graph* is a graph with more that two vertices with no separation pairs. In a triconnected graph there are at least three vertex disjoint paths between every pair of non adjacent vertices.

Let $\{a, b\}$ be a separation pair in the graph G that induces the separation classes $E_1, E_2, \ldots, E_n$. Assume $1 \leq m < n$ and let $E' = \bigcup_{i=1}^{m} E_i$ and $E'' = \bigcup_{i=m+1}^{n} E_i$ such that $|E'| \geq 2$ and $|E''| \geq 2$. Then we will refer to the graphs $G' = (V(E'), E')$ and $G'' = (V(E''), E'')$ as the *separating graphs* of G. The graphs

$$G_1 = (V(E'), E' \cup \{(a, b)\})$$

and

$$G_2 = (V(E''), E'' \cup \{(a, b)\})$$

are called *split graphs* of G. The added edge (a, b) is labeled to denote the split and is called a *virtual edge.* Assume that the graph G and its split graphs are recursively split until obtaining graphs that cannot be split further. The set of these graphs defines the set of *split components* of G. Note that the split components are triconnected graphs and that by merging the split components we recover the original graph.

Hopcroft and Tarjan, [28], and Miller and Ramachandran, [48], reported on algorithms to efficiently compute separating graphs and split components of a graph. Most concepts in these algorithms were first reported in the planarity testing algorithm of Hopcroft and Tarjan, see Even's book [17].

3.1.2 Geometric Constraint Graphs

Abstract problems can be represented by using a graph in a straightforward manner. Those graphs are named *geometric constraint graphs.* Geometric constraint graphs are simple and undirected.

Let $A = \langle G, C, P \rangle$ be an abstract problem. The associated geometric constraint graph $G = (V, E)$ is a simple, undirected graph such that:

- Every vertex corresponds to a geometric element in the abstract problem.
- Every edge corresponds to a constraint in the abstract problem. Assume that e is an edge and $r(g_1, g_2, p_1, \ldots, p_n)$ its corresponding constraint where g_i are geometric elements and p_i are parameters. Then e is incident to vertices g_1 and g_2 in the graph.

Example 3.1 Let us consider the following geometric problem corresponding to the sketch in Figure 2.1 (page 8):

```
gcp A
  param
    $d_1, d_2, a_1, a_2, h_1$ : real
  endparam
  geom
    $p_1, p_2, p_3, p_4$ : point
    $l_1, l_2, l_3, l_4$ : line
  endgeom
  $onPL(p_1, l_1)$
  $onPL(p_1, l_2)$
  $onPL(p_2, l_1)$
  $onPL(p_2, l_3)$
  $onPL(p_3, l_3)$
  $onPL(p_3, l_4)$
  $onPL(p_4, l_4)$
  $onPL(p_4, l_2)$
  $distPP(p_2, p_3, d_1)$
  $distPP(p_3, p_4, d_2)$
```

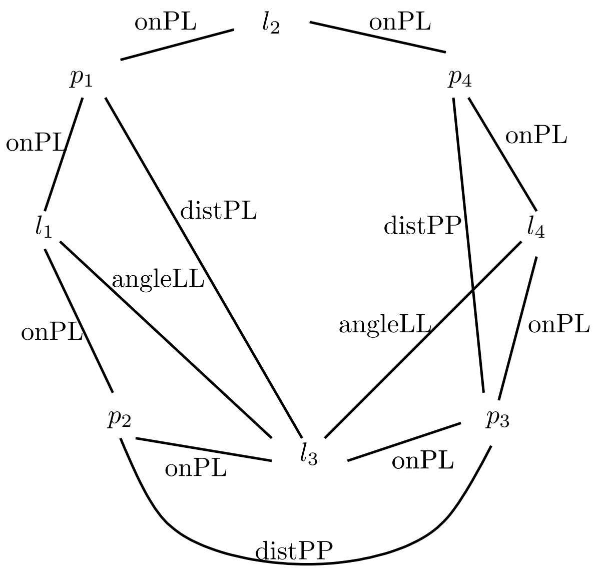


Figure 3.1: Geometric constraint graph corresponding to the abstract problem A.

$distPL(p_1, l_3, h_1)$
$angleLL(l_3, l_1, a_2)$
$angleLL(l_3, l_4, a_1)$
endgcp

The associated geometric constraint graph is defined by a set of vertices $V = \{p_1, p_2.p_3, p_4, l_1, l_2, l_3, l_4\}$ and a set of 13 edges corresponding to every constraint in the abstract problem. For instance, there is an edge between p_1 and l_1 corresponding to the constraint $onPL(p_1, l_1)$. Figure 3.1 shows the graph.

$\diamondsuit$

Representing an abstract problem as a graph imposes some limitations on the constraint language $\mathcal{L}_R$. Because edges in a graph are incident to two vertices, all predicates in $\mathcal{L}_R$ must be defined on two geometric elements and any number of parameters.

3.1.3 Well Constrained Graphs

In Chapter 2 we have defined the abstract problem concept. However, not all the abstract problems that are syntactically correct make sense. For instance, Figure 3.2 shows an abstract problem that does not define any

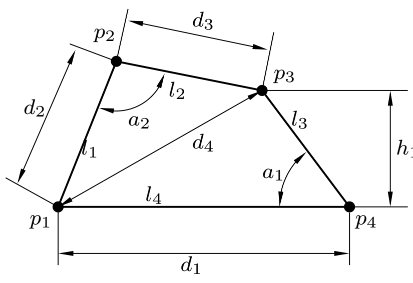


Figure 3.2: Over constrained abstract problem

realization for most of the parameters assignments. The reason is that the number of constraints involved is larger than required. We are interested in abstract problems that make sense. Roughly speaking, we say that an abstract problem that makes sense is well constrained. The aim of this Section is to further understand the concept of well constrained.

Let us focus first on instance geometric constraint problems. An instance problem results from applying a parameters substitution to an abstract problem. Assume that A is an abstract problem and α a parameters assignment for A. Then, the set of realizations of $\alpha.A$, $V(\alpha.A)$, is either:

1. An empty set. In this case we say that the instance problem is *over constrained.*

2. An infinite set. In this case we say that the instance problem is *under constrained.*

3. A non empty finite set. In this case we say that the instance problem is *well constrained.*

Notice that an instance problem can also be represented by a system of equations. These are the equations corresponding to the characteristic formula of the instance problem. Then, the set of solutions to these equations naturally defines when an instance problem is well, under or over constrained.

Our interest is centered on abstract problems. Now the question is: when are abstract problems well defined? A first attempt to answer the question may lead to say that an abstract problem is well (over, under) constrained when all the instance problems derived from it are well (over, under) constrained.

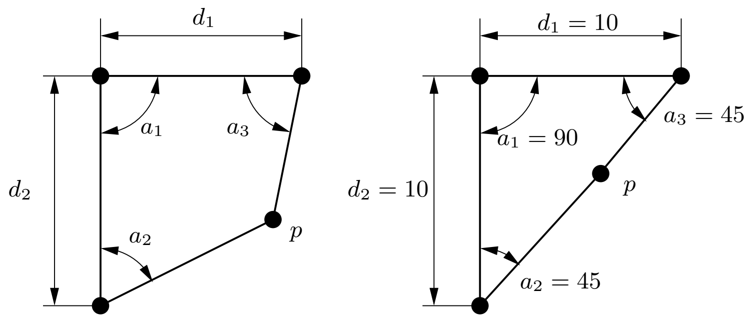


Figure 3.3: Well constrained problem and a degenerate instance.

Nonetheless, things are more difficult. Some particular examples illustrate these difficulties.

1. There are well constrained abstract problems that, with some parameter assignments, become under constrained. Figure 3.3, due to Bouma, [4], illustrates a well constrained abstract problem that becomes under constrained for a specific parameter assignment. In the sketch on the left there is a well constrained problem. If the parameter assignment of the sketch on the right is applied, then the position of point p becomes undefined and the problem has an infinite number of realizations.

2. Over constrained abstract problems do not define any realization. However, there are particular parameter assignments for which an over constrained problem defines a finite set of realizations. In the example of Figure 3.4, all parameter assignments such that $d_3^2 = d_1^2 + d_2^2 - (2d_1 d_2 \cos a_1)$ define realizations.

The definition of well constrained abstract problems has been studied in the context of rigidity theory. Whiteley in [63, 64] describes some of the main results. The book of Graver *et al.*, [23], includes a comprehensive work on this topic. Finally, Owen in [51] collects some of the main results.

Rigidity theory uses the concept of *generically well constrained* to denote an abstract system which is well defined. Roughly speaking, we say that an abstract problem is generically well constrained if it is well constrained for most of the parameter assignments except for the degenerate ones. We refer

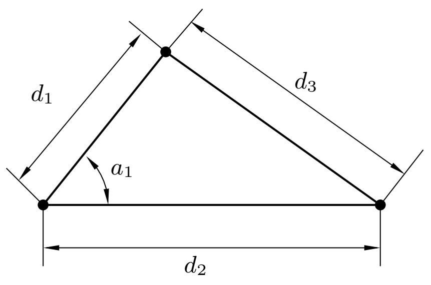


Figure 3.4: Over constrained problem which has some degenerated realizations.

to the introduction of Graver's book, [23], for an excellent tutorial on this topic.

To make the concept of generically well constrained useful we need properties that characterize which abstract problems are generically well constrained or, equivalently, which geometric constraint graphs are generically well constrained. Intuitively, this depends on the graph properties. Several results, according to the geometric elements and the kind of constraints found in the graph, have been reported. The first result characterizes geometric constraint graphs associated with geometric constraint problems including only points and distances. It is due to Laman, [43].

Theorem 3.1 (Laman) *Let $G = (V, E)$ be a geometric constraint graph with $|V| \geq 2$ such that the geometric elements are points and the constraints are distances between points. Then G is a* generically well constrained *graph iff*

1. *for every set of geometric elements $V' \subseteq V$, the induced subgraph $G' = (V', E')$ has the following property:*

$$|E'| \leq 2|V'| - 3$$

2. $|E| = 2|V| - 3$

In this Theorem, the first condition is usually known as the *Laman condition.*

There are other results which apply to distinct sets of geometric elements and constraints. Servatius and Whiteley, [56], give a characterization of generically well constrained graphs with points and directed segments in which constraints are point to point distances and segment orientations. Owen and Whiteley, [52], present the characterization of generically well constrained graphs with points and lines in which constraints are distances between geometrical elements of any type.

Because our interest focus on the constructive geometric constraint solving, we are interested in a particular kind of graphs. These graphs include geometric elements as points and lines related by constraints like distance between any pair of elements and angle between lines. Unfortunately, we do not know of any results for this kind of graphs. The best approximation is due to Owen, [51], who gives a necessary condition for the case where angle constraints between lines are also allowed. Nevertheless, no sufficient condition is known in this case. The result is the following:

Theorem 3.2 *Let $G = (V, E)$ be a geometric constraint graph with $|V| \geq 2$ such that the geometric elements are points or lines and the constraints are angle between two lines, distances between two points and distance between a point and a line. If G is a generically well constrained problem then,*

1. *for every set of geometric elements $V' \subseteq V$, the induced subgraph $G' = (V', E')$ has the following property:*
$$|E'| \leq 2|V'| - 3$$
2. *for every set of lines in the graph $L' \subseteq \{v \in V | v \text{ is a line}\}$, the induced graph $G' = (L', E')$ has the following property:*
$$|E'| \leq |L'| - 1$$

An example showing that this theorem grants only a necessary condition is given in Owen, [51], and it is reproduced in Figure 3.5. In this figure vertices labeled p are points, and vertices labeled l are lines. Although this graph fulfills the conditions of Theorem 3.2 it is over constrained.

The lack of a complete characterization of generically well constrained graphs leads Fudos, [19], and Soto, [58], to define the concepts of structurally well, under and over constrained. These concepts classify the geometric constraint graphs only paying attention to their structural characteristics and ignoring the type of the geometric elements and constraints represented by nodes and edges, respectively.

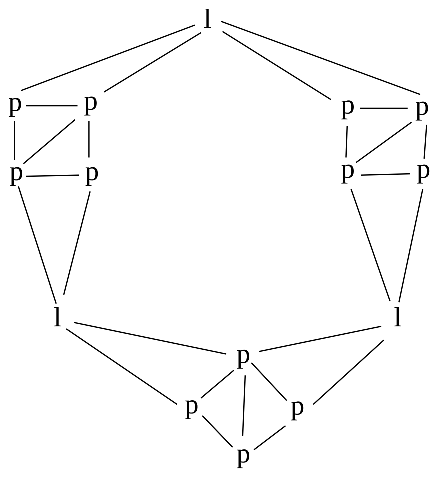


Figure 3.5: Over constrained graph for which Theorem 3.2 holds.

Definition 3.3 *Let A be an abstract problem. Let $G = (V, E)$ be the graph form of A with $|V| \geq 2$.*

A is structurally over constrained *if there is an induced subgraph of G with $m < |V|$ vertices such that the number of edges is greater than $2m - 3$.*

A is structurally under constrained *if it is not structurally over constrained and the number of constraints is less than $2|V| - 3$.*

A is structurally well constrained *if it is not structurally over constrained and the number of constraints is equal to $2|V| - 3$.*

This definition is used extensively throughout this manuscript as a way to refer to generically well constrained problems. It should be understood that this definition classifies graphs only by their structural properties. Moreover, it does not distinguish between graphs with the same structure and different geometric elements or constraints. Therefore, there are graphs which are structurally well constrained and not generically well constrained. The Figure 3.5 shows one of them.

In the rest of the manuscript we will use the term well (under, over) constrained to refer to structurally well (under, over) constrained property. When needed, we will conveniently qualify the term as structurally or generically well (under, over) constrained.

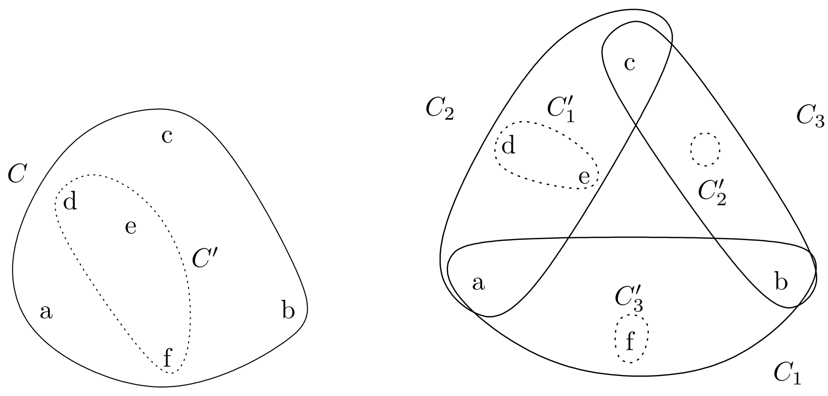


Figure 3.6: Left: A set C. Right. A set decomposition of C.

3.1.4 Set Decomposition of a Graph

In this section first we define the concept of *set decomposition* that refers to a way of partitioning a given abstract set. Then we define the concept of *set decomposition of a graph.* This tool will be used in the following sections to describe Fudos and Hoffmann's decomposition and reduction analysis algorithms.

Definition 3.4 *Let C be a set with, at least, three different members, say a, b, c. Let $\{C_1, C_2, C_3\}$ be three subsets of C. We say that $\{C_1, C_2, C_3\}$ is a* set decomposition *of C if*

1. $C_1 \cup C_2 \cup C_3 = C$*,*

2. $C_1 \cap C_2 = \{a\}$*,*

3. $C_1 \cap C_3 = \{b\}$ *and*

4. $C_2 \cap C_3 = \{c\}$*.*

We say that $\{a, b, c\}$ are the active elements *of the set decomposition.*

Figure 3.6 shows a set and a possible set decomposition. Next we define the concept of set decomposition of a graph, illustrated in Figure 3.7.

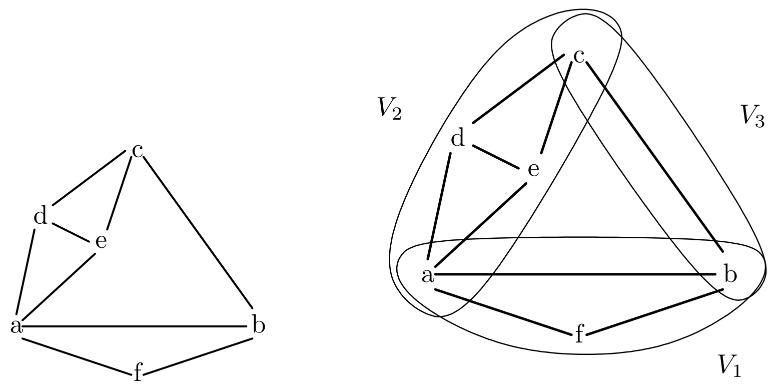


Figure 3.7: Left: Graph. Right: Set decomposition of the graph.

Definition 3.5 *Let $G = (V, E)$ be a graph. Let $\{V_1, V_2, V_3\}$ be three subsets of V. Then $\{V_1, V_2, V_3\}$ is a* set decomposition *of G if it is a set decomposition of V and for every edge e in E, $V(e) \subseteq V_i$ for some i, $1 \leq i \leq 3$.*

Roughly speaking, a set decomposition of a graph $G = (V, E)$, is a set decomposition of the set of vertices V such that does not *break* any edge in E. Figure 3.8 left shows a graph $G = (V, E)$ and Figure 3.8 right shows a set decomposition of V which is not a set decomposition of G because vertices incident to edge (e, b) do not belong to the same set in the partition.

The set decomposition and the separation pairs of a graph are related by the following lemma.

Lemma 3.6 *Let $\{V_1, V_2, V_3\}$ be a set decomposition of a graph G and let $V_1 \cap V_2 = \{a\}$ and $V_1 \cap V_3 = \{b\}$. If $|V_1| > 2$, then $\{a, b\}$ is a separation pair of G.*

Proof
The subgraphs of G induced by V_i, for $1 \leq i \leq 3$, have disjoint sets of edges. By Definition 3.4 $V_1 \cap (V_2 \cup V_3) = \{a, b\}$. Thus, removing $\{a, b\}$ disconnects G. Therefore $\{a, b\}$ is a separation pair. $\square$

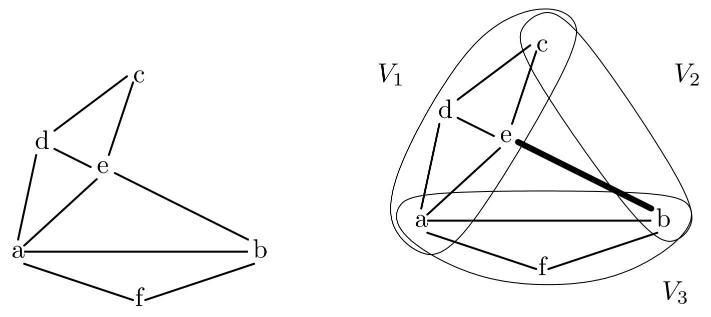


Figure 3.8: Left: Graph. Right: Set decomposition with a broken edge.

3.2 Fudos and Hoffmann's Reduction Algorithm

We recall the formalization of the reduction analysis method proposed by Fudos and Hoffman in [20, 21]. This method solves a geometric constraint problem by analyzing the constraint graph bottom-up. *Clusters* play a central role in this method. A *cluster* is a set of two dimensional geometric elements with known positions with respect to a local coordinate system, i.e. their relative positions are known. An *elemental cluster* is a cluster with exactly two geometric elements and one geometric constraint between them. Figure 3.9 shows the different elemental clusters that can be defined with lines and points.

In an elemental cluster, the relative position of one geometric element with respect to the other is fixed. Moreover, three clusters that pairwise share an element define a rigid body and can be merged into one new cluster whose elements are the union of the elements in the merged cluster. See Figure 3.10.

The Fudos and Hoffman reduction algorithm can be described in terms of abstract reduction systems as follows. See [42] for the terminology and notation on this topic.

Given a geometric constraint graph $G = (V, E)$, we define the *initial set of clusters* $\mathbf{S}_G$ by $\mathbf{S}_G = \{\{u, v\} \mid (u, v) \in E\}$. Let $\mathbf{S}$ be a set of clusters in which there are three clusters C_1, C_2, C_3 such that $\{C_1, C_2, C_3\}$ is a set

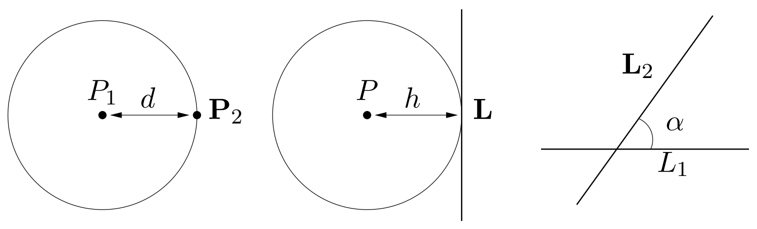


Figure 3.9: Elemental clusters built from points and lines.

decomposition of C. Then, $\mathbf{S} \longrightarrow_r \mathbf{S}'$ is a reduction rule where

$$\mathbf{S}' = (\mathbf{S} - \{C_1, C_2, C_3\}) \cup C.$$

The geometric constraint problem represented by the geometric constraint graph G is *solvable by reduction analysis* if $\mathbf{S}_G$ reduces to the singleton $\{V\}$.

Let $\longrightarrow_r^*$ be the transitive and reflexive closure of the reduction relation $\longrightarrow_r$ and let $S(G) = \{\mathbf{S} \mid \mathbf{S}_G \longrightarrow_r^* \mathbf{S}\}$. Fudos and Hoffmann proved in [20] that if G is not over constrained, the abstract reduction system $\mathcal{R} = \langle S(G), \longrightarrow_r \rangle$ is terminating and confluent which implies the unique normal form property and canonicity. In particular, it means that no matter the reduction sequence we choose, we finally get the same set of clusters.

Fudos and Hoffmann presented in [21] an algorithm that efficiently implements the reduction analysis. This algorithm has a quadratic worst case running time on the number of geometric elements of the problem.

3.3 Fudos and Hoffmann's Decomposition Algorithm

Fudos in [19] and Fudos and Hoffmann in [20, 21] propose an algorithm based on the decomposition of a set of clusters. In this section we recall this algorithm. The algorithm was developed to be run once the reduction algorithm explained in the previous section has computed a partial solution of a geometric constraint graph. However, it can also be applied to a completely unsolved graph and thus the algorithm constitutes an analysis method on its own. Here we are interested in this second application of the algorithm.

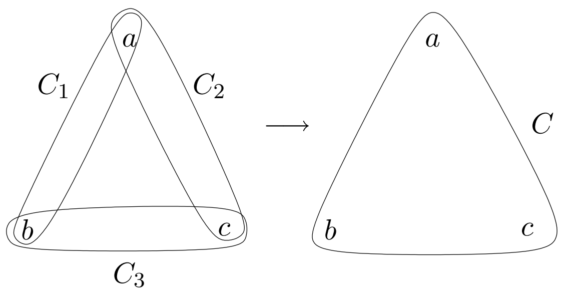


Figure 3.10: Cluster merging

The input of the algorithm is a geometric constraint graph and the output is the sequence of rules (decompositions) which have been applied by the algorithm. The algorithm has an efficient version whose worst case running time is $O(|V|^2)$. We describe this algorithm using an abstract reduction system.

Given a geometric constraint graph $G = (V, E)$, define the initial set of clusters $\mathbf{O}_G = \{V\}$. Let $\mathbf{O}$ be a set of clusters including a cluster C such that $\{C_1, C_2, C_3\}$ is a set decomposition of the subgraph of G induced by C. Then, $\mathbf{O} \longrightarrow_d \mathbf{O'}$ is a reduction rule where

$$\mathbf{O'} = (\mathbf{O} - C) \cup \{C_1, C_2, C_3\}.$$

Example 3.2 Assume that we apply the decomposition algorithm to the graph of the Figure 3.1. The initial set of clusters is $\mathbf{O}_G = \{\{p_1, p_2.p_3, p_4, l_1, l_2, l_3, l_4\}\}$. The reduction rule must compute a set decomposition $\{C_1, C_2, C_2\}$. Let $C_1 = \{l_2, p_1\}$, $C_2 = \{l_2, p_4\}$ and $C_3 = \{p_2, p_3, l_1, l_3, l_4\}$ be this set decomposition. Then this is a decomposition of $\mathbf{O}_G$ that can be written:

$$\mathbf{O}_G \longrightarrow_d \mathbf{O}_G^1 = \{\{l_2, p_1\}, \{l_2, p_4\}, \{p_2, p_3, l_1, l_3, l_4\}\}$$

Figure 3.11 illustrates the application of this reduction rule.

$\Diamond$

The geometric constraint problem represented by the geometric constraint graph G is *solvable by decomposition analysis* if $\mathbf{O}_G$ reduces to $\mathbf{S}_G = \{\{u, v\} \mid (u, v) \in E(G)\}$. The output of the algorithm is the sequence of reductions applied.

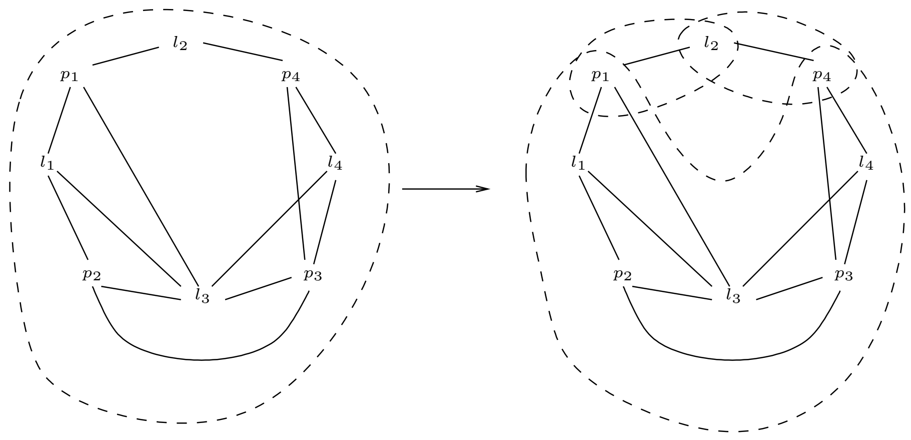


Figure 3.11: Application of the reduction rule in a decomposition algorithm.

Let $\longrightarrow_d^*$ be the transitive and reflexive closure of the reduction relation $\longrightarrow_d$ and let $O(G) = \{\mathbf{O} \mid \mathbf{O}_G \longrightarrow_d^* \mathbf{O}\}$. Then the decomposition analysis can be seen as the abstract reduction system $\mathcal{D} = \langle O(G), \longrightarrow_d \rangle$, [42].

3.4 Owen's Decomposition Algorithm

First we briefly recall the Owen's algorithm. Then we give a new formalization for the Owen's algorithm. The new formalization is simpler and will be used in the rest of the manuscript.

3.4.1 Owen's Algorithm

Owen in [50] introduced a geometric constraint solving technique based on a top-down analysis of the geometric constraint graph associated with a geometric problem.

The algorithm has two steps. In a first step, the algorithm computes the set of split components S of the given graph G, [28]. These split components are either triangles or complex triconnected graphs, that is, graphs with more than three edges. As computed, the complex split components are no further decomposable. To overcome this problem, in a second step the complex split components are transformed, if possible, by removing from

```
func Owen($G$)
  $SC$ := SplitComponents($G$)
  $S := \emptyset$
  foreach $g$ in $SC$ do
    if Reducible($g$) then
      $S := S \cup$ Owen(Reduce($g$))
    else
      $S := S \cup \{g\}$
    fi
  done
  return $S$
end
```

Figure 3.12: Owen's analysis algorithm.

the graph one of the virtual edges introduced in the first step. Note that virtual edges are always incident to separation pairs.

Then the first step is recursively applied to the transformed split components. The algorithm terminates when the graphs cannot be split further.

At the end of the analysis, the original graph has been decomposed into a set of triangles whose edges are either original edges or virtual edges.

If function `SplitComponents(G)` computes the split components of G, function `Reducible(g)` checks whether a split component should be further subdivided, and function `Reduce(g)` removes from graph g unneeded virtual edges, Owen's algorithm can be written as shown in Figure 3.12.

How Owen's analysis algorithm works is illustrated in Figure 3.13. Virtual edges are shown in dashed lines.

3.4.2 The New Formalization

To decompose a graph, Owen's method uses the algorithm for finding triconnected components reported by Hopcroft and Tarjan in [28], which is based on preserving graph connectivity. As a result, the split components generated by the decomposition include extra virtual edges. To recursively apply the decomposition process, Owen's algorithm must remove these extra

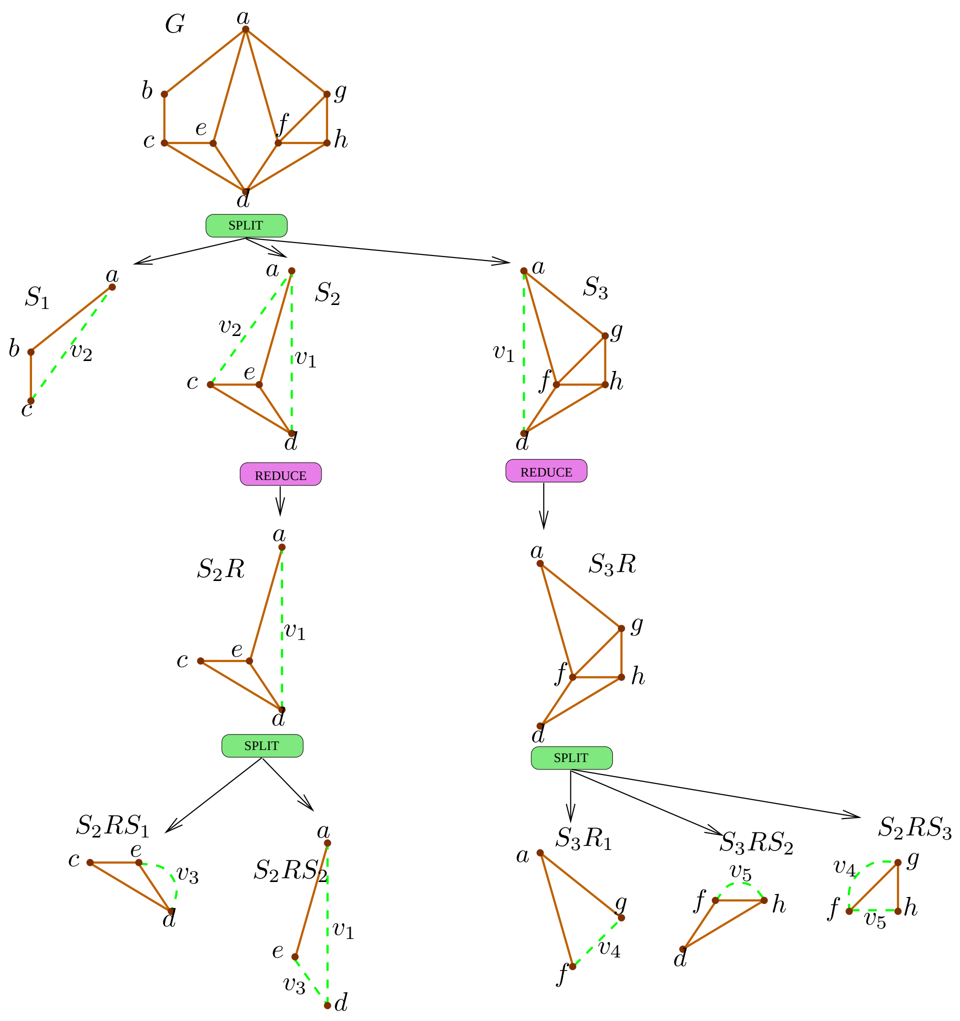


Figure 3.13: Owen's algorithm computation applied to an example graph.

virtual edges.

In what follows we will present an algorithm to decompose a constraint graph in triconnected graphs with exactly three vertices, that is, triangles. The algorithm is based on a divide and conquer strategy which preserves the constraint graph property of being well constrained. The resulting algorithm is conceptually simple and easy to implement.

Similarly to the techniques reported in [21, 50], our algorithm will be based on subdividing the constraint graph into two separating graphs induced by a separation pair. With the aim of clearly stating a subdivision criterion, we start by giving some definitions and deriving properties which relate well constrained graphs with their separating graphs.

Definition 3.7 *Let $G = (V, E)$ be a geometric constraint graph. We define the* `Deficit` *function associated with G by*

$$\mathit{Deficit}(G) = (2|V| - 3) - |E|$$

The function `Deficit` computes the difference between the number of edges needed for a constraint graph to be well constrained and its actual number of edges. Note that if G is not over constrained, $\text{Deficit}(G) \geq 0$.

Lemma 3.8 *Let G be a constraint graph and G' and G'' separating graphs. Then*

$$\mathit{Deficit}(G) = \mathit{Deficit}(G') + \mathit{Deficit}(G'') - 1$$

Proof
By definition, $\text{Deficit}(G) = (2|V| - 3) - |E|)$. Since G' and G'' are separating graphs of G, then $|V| = |V'| + |V''| - 2$ and $|E| = |E'| + |E''|$. Therefore,

$$\begin{aligned}\text{Deficit}(G) &= 2(|V'| + |V''| - 2) - 3 - (|E'| + |E''|) \\ &= (2|V'| - 3 - |E'|) + (2|V''| - 3 - |E''|) - 1 \\ &= \text{Deficit}(G') + \text{Deficit}(G'') - 1\end{aligned}$$

□

Lemma 3.9 *Let G be a well constrained graph and G' and G'' separating graphs. Then if Deficit(G') > Deficit(G''), G' is under constrained and G'' is well constrained.*

Proof
Since G is well constrained, Deficit(G) = 0 and separating graphs, G' and G'', are not over constrained, that is, Deficit(G') ≥ 0 and Deficit(G'') ≥ 0. From Lemma 3.8, Deficit(G) = Deficit(G') + Deficit(G'') $- 1$. Thus Deficit(G') + Deficit(G'') = 1. Then, Deficit(G') = 1 and Deficit(G'') = 0, which means that G' is under constrained and G'' well constrained. □

Definition 3.10 *Let G be a constraint graph. Let G' and G'' separating graphs of G. The* modified split graphs*, G_1 and G_2, of G are defined as follows. If Deficit(G') > Deficit(G'') then*

$$G_1 = (V(E'), E' \cup \{(a,b)\}) \quad \textit{and} \quad G_2 = G''$$

Lemma 3.11 *Let $G = (V, E)$ be a constraint graph and, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be modified split graphs. Then Deficit(G) = Deficit(G_1) + Deficit(G_2).*

Proof
Now $|E| = |E_1| + |E_2| - 1$. Apply proof of Lemma 3.8. □

Definition 3.12 *Let G be a geometric constraint graph. An* s-tree *S of G is a binary tree of graphs such that:*

1. *the root is the graph G,*
2. *for each node G' in S its subtrees are rooted in the modified split graphs G'_1 and G'_2 of G', and*
3. *the leaves are either triangles or triconnected graphs.*

As we will see in Section 4.3 we are interested in graphs for which there are s-trees whose leaf nodes are triangles because we know how to solve the associated geometric constraint problem,[50].

Definition 3.13 *We say that a constraint graph G is* s-tree decomposable *if there is an s-tree such that its root is G and all its leaves are triangles.*

Let `Triconnected(`G`)` be a function that tests whether a graph has a separation pair, `SeparatingGraphs(`G`)` a function that computes the separating graphs of G, (Recall that separating graphs do not include virtual edges),

```
func Analysis($G$)
  if Triconnected($G$) then
    $S$ := BinaryTree($G$, nullTree, nullTree)
  else
    $G_1$,$G_2$ := SeparatingGraphs($G$)
    if Deficit($G_1$) > Deficit($G_2$) then
      $G_1$ := AddVirtualEdge($G_1$)
    else
      $G_2$ := AddVirtualEdge($G_2$)
    fi
    $S$ := BinaryTree($G$, Analysis($G_1$),
                     Analysis($G_2$))
  fi
  return $S$
end
```

Figure 3.14: New algorithm for decomposition analysis.

and `AddVirtualEdge(`G`)` a function that adds a virtual edge incident to the separation pair used to compute the split graph G. Then the decomposition analysis algorithm based on preserving deficits of graphs can be written as shown in Figure 3.14.

The input to the algorithm is a graph G associated to a geometric constraint problem. The output is an s-tree S whose root is G. Note that if G is *s-tree decomposable* the resulting s-tree decomposes G into triangles and the problem is solved.

Figure 3.15 illustrates the behavior of the new decomposition analysis algorithm applied to the example graph in Figure 3.13. Note that now only those virtual edges that are strictly necessary to keep the deficit property are included in the modified split graphs, therefore avoiding the need for graph transformation.

When one of the separation classes is a single edge, it is incident to the vertices in the separation pair. In this case we prove the following result.

Lemma 3.14 *Let $G = (V, E)$ be a well constrained geometric constraint*

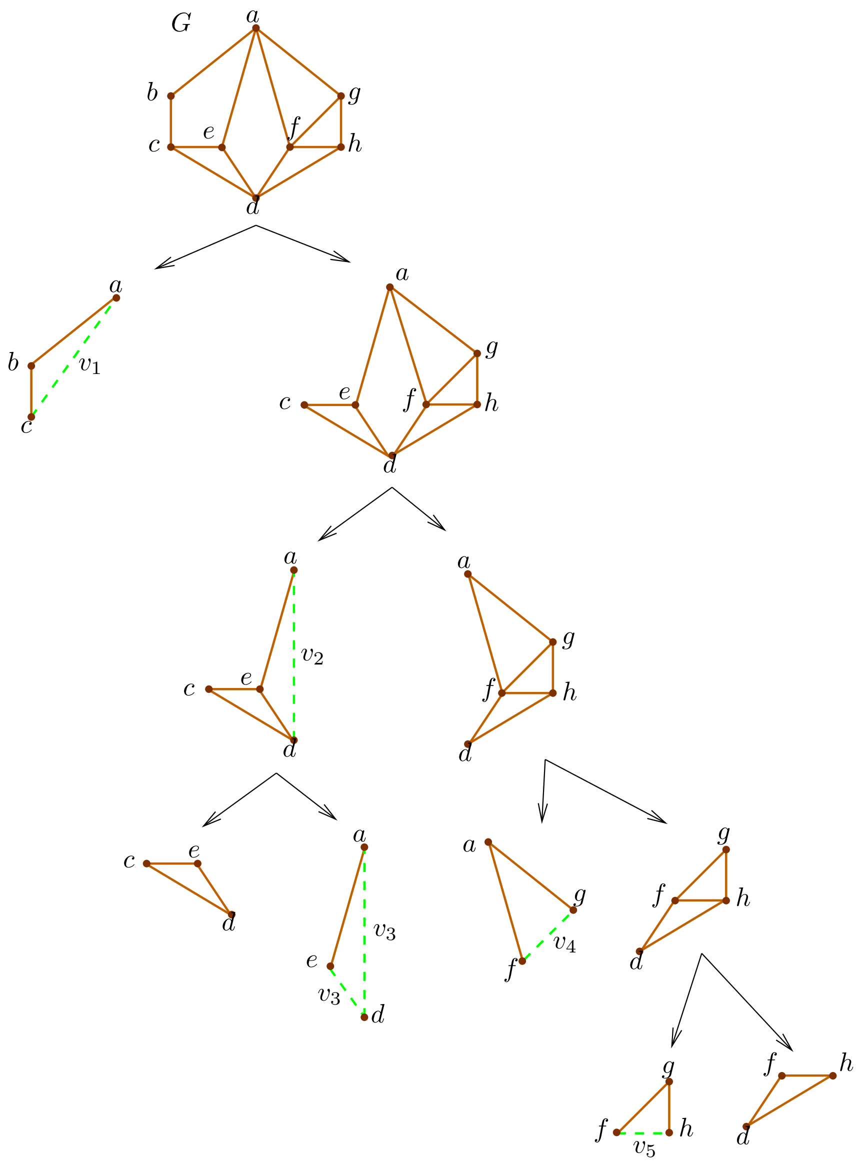


Figure 3.15: Decomposition analysis generated by the new algorithm on the example graph in Figure 3.13.

graph and $\{a,b\}$ a separation pair such that $(a,b) \in E$. Then the separation graph which contains edge (a,b) is well constrained.

Proof

Let $\{a,b\}$ be the separation pair in the graph $G = (V,E)$ and $E_1,\ldots,E_{n-1},E_n$ be the separation classes, where E_n contains just the edge (a,b). Let $\mathcal{E}' = \bigcup_{i=1}^{m} E_i$ and $\mathcal{E}'' = \bigcup_{i=m+1}^{n-1} E_i$ such that $|\mathcal{E}'| \geq 2$ and $|\mathcal{E}''| \geq 2$. We have

$$|E| = |\mathcal{E}'| + |\mathcal{E}''| + |E_n| = |\mathcal{E}'| + |\mathcal{E}''| + 1$$

$$|V| = |V(\mathcal{E}')| + |V(\mathcal{E}'')| - 2$$

G well constrained means that $2|V| - 3 - |E| = 0$. Substituting $|E|$ and $|V|$ by the expressions above and rearranging terms

$$(2|V(\mathcal{E}')| - 3 - |\mathcal{E}'|) + (2|V(\mathcal{E}'')| - 3 - |\mathcal{E}''|) - 2 = 0$$

There are two different cases. First let

$$(2|V(\mathcal{E}')| - 3 - |\mathcal{E}'|) = (2|V(\mathcal{E}'')| - 3 - |\mathcal{E}''|) = 1$$

Since classes E_i are grouped arbitrarily, assume that $E' = \mathcal{E}' \cup E_n$. Then $V(E') = V(\mathcal{E}')$ and $|E'| = |\mathcal{E}'| + 1$. Then the separation graphs are $G' = (V(E'), E')$ and $G'' = (V(\mathcal{E}''), \mathcal{E}'')$. Thus the deficit of the separation graph $G' = (V(E'), E')$ is

$$\begin{aligned}(2|V(E')| - 3 - |E'|) &= 2|V(\mathcal{E}')| - 3 - (|\mathcal{E}'| + 1)\\ &= 2|V(\mathcal{E}')| - 3 - |\mathcal{E}'| - 1\\ &= 0\end{aligned}$$

Therefore the separation graph G' contains edge (a,b), is well constrained and, since Deficit$(G') <$ Deficit(G''), no virtual edge is added to it.

The second case leads to a contradiction. Without loss of generality, let $(2|V(\mathcal{E}')| - 3 - |\mathcal{E}'|) = 0$, that is, G' is well constrained. Let $(2|V(\mathcal{E}'')| - 3 - |\mathcal{E}''|) = 2$ and assume again $E' = \mathcal{E}' \cup E_n$. This would result in the separation graph $G' \subset G$ being over constrained, that is G would be over constrained which is a contradiction. $\square$

Lemmas 3.8, 3.9 and 3.11 along with Lemma 3.14 prove that the algorithm preserves the deficit of the input graph.

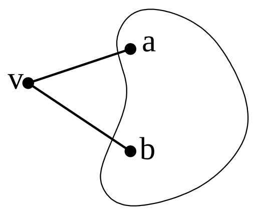


Figure 3.16: Graph with a degree two vertex v.

3.4.3 Subdivision Pattern

The algorithm given in the previous section analyzes a constraint graph by decomposing it into two split graphs induced by a separation pair. However, there is nothing essential in this subdivision method.

Todd, [59], reported on a method where graphs are subdivided by isolating vertices of degree two from their neighbors. In fact, this is a particular case of decomposing through separation pairs. Figure 3.16 illustrates a graph with a degree two vertex v. Note that its neighbors, a and b, are a separation pair. This subdivision method is rather limited but can be satisfactorily combined with other methods, like those explained by Hopcroft *et al.*, [28], or Miller *et al.*, [48], to compute more general graph subdivisions.

Another method subdivides a graph into three subgraphs by selecting three vertices such that by removing them the graph splits into three connected components. See Figure 3.17. We do not know of any efficient algorithm to select the three vertices but they can always be computed by using a brute force approach.

3.5 Summary

In this chapter we have introduced the geometric constraint graphs as a tool to represent geometric constraint abstract problems. Geometric constraint graphs are the representation used for the analysis algorithms studied in this Chapter.

We have also discussed the characterization of well constrained geometric constraint graphs. We have illustrated the difficulty of defining this concept. Finally we show the relationship between this concept and the generically

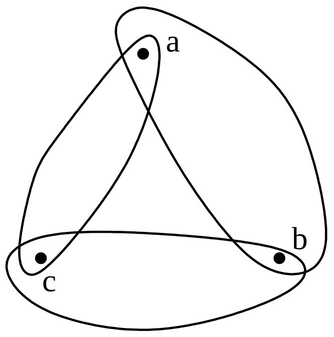


Figure 3.17: Subdividing a graph into three subgraphs by using three vertices a, b and c.

well constrained concept from the rigidity theory. We have introduced the definition of structurally well constrained graphs which is extensively used throughout all the manuscript.

After that, we have reviewed three analysis methods: decomposition, reduction, and Owen's method. The three methods are conveniently reformulated to easily reach the objectives of the next chapter. Decomposition and reduction have been reformulated as abstract reduction systems with the help of the set decomposition of a graph. Owen's method have been reformulated applying the concept of deficit and a new algorithm has been devised from this reformulation. We proved the correctness of the new algorithm. Moreover, the new formalization reveals that Owen's algorithm is a plain divide-and-conquer algorithm that preserves the deficit value while splitting the problem.

4 Domain of Analysis Methods

In this Chapter we prove that the three analysis methods studied in Chapter 3 have the same domain. We first define the tree decomposition of a graph and prove some properties. Then, the class of full tree decomposable geometric constraint graphs is defined. We show that the domain of each studied method is equivalent to the class of full tree decomposable graphs. Finally we show how this class of graphs can be defined as a recursive sequence similar to Henneberg sequences of well constrained graphs, see Graver [23].

Most of the work in this chapter has been previously published in [34, 35, 37, 39].

4.1 Tree Decomposition of a Graph

In this section, we define the concept of *tree decomposition* of a graph. This concept mimics the data structure obtained by recursively applying a set decomposition to a geometric constraint graph.

Definition 4.1 *Let $G = (V, E)$ be a graph. A 3-ary tree T is a* tree decomposition *of G if*

1. *V is the root of T,*
2. *Each node $V' \subseteq V$ of T is the father of exactly three nodes, say $\{V'_1, V'_2, V'_3\}$, which are a set decomposition of the subgraph of G induced by V', and*

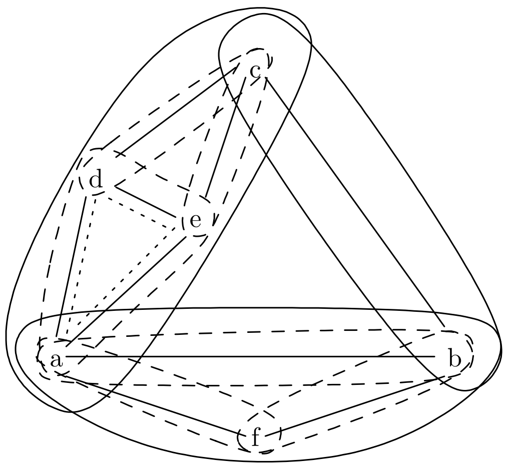


Figure 4.1: Collection of set decompositions of the graph in Figure 3.7.

3. *Each leaf node contains exactly two vertices of V.*

A graph for which there is a tree decomposition is a *tree decomposable* graph. Figure 4.1 shows a collection of set decompositions recursively generated for the tree decomposable graph of Figure 3.7. The corresponding tree decomposition is shown in Figure 4.2.

By Definition 4.1, all leaves of a tree decomposition T of a graph G have cardinality two. Moreover, in a tree decomposable graph every edge is mapped to a tree decomposition leaf. We prove this in the following result:

Theorem 4.2 *Let $G = (V, E)$ be a geometric constraint graph and T be a tree decomposition of G. For each edge $(a, b) \in E$, there is a leaf $\{a, b\}$ in T.*

Proof

Let us proceed by induction on the structure of T, see the book by Manna, [45].

Base case: T is a tree with only the root $V = \{a, b\}$. T corresponds to a graph $G = (V, E)$ and E is either the empty set or a singleton containing the edge (a, b). In both cases the proposition holds.

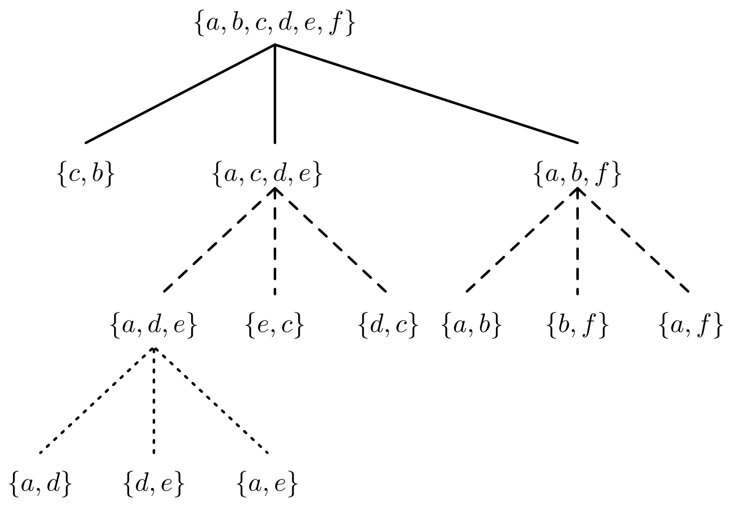


Figure 4.2: Tree decomposition of the graph in Figure 4.1.

Induction hypothesis: For all T', a subtree of T with root V', let $G' = (V', E')$ be the subgraph of G induced by V'. For each edge $(a,b) \in E'$, there is a leaf $\{a,b\}$ in T'.

Induction step: Let T be a tree decomposition of G. V is the root of T and let V_i be the root of tree decomposition T_i, the i-th child of V, for $1 \leq i \leq 3$. Let $G_i = (V_i, E_i)$ be the subgraph of G induced by V_i, for $1 \leq i \leq 3$. $\{V_1, V_2, V_3\}$ is a set decomposition of G. Since a set decomposition of a graph induces a partition on the set of edges of the graph, G_i for $1 \leq i \leq 3$ are edge-disjoint graphs. By induction hypothesis, for each edge $(a,b) \in E_i$, there is a leaf $\{a,b\}$ in T_i, for $1 \leq i \leq 3$. Moreover, the leaves of T are the leaves of all T_i and $E = E_1 \cup E_2 \cup E_3$. Thus, for each edge $(a,b) \in E$, there is a leaf $\{a,b\}$ in T. $\square$

There are graphs for which a tree decomposition exists such that there is a one to one correspondence between the leaves of the tree and the edges of the graph. This suggests the following definition:

Definition 4.3 *Let $G = (V,E)$ be a geometric constraint graph. T, a tree decomposition of G, is a* full tree decomposition *of G if there is a one-to-one correspondence between the leaves of T and the edges of G.*

Notice that although in a tree decomposable graph every edge is mapped to a tree decomposition leaf, the converse is only true for full tree decomposable graphs.

A tree decomposable graph can not be over constrained. The following result proves this property.

Theorem 4.4 *Let $G = (V, E)$ be a geometric constraint graph. If G is tree decomposable, then G is not over constrained.*

Proof
We proceed by induction on the number of vertices of G.

Base case: $|V| = 2$. Assume that G is tree decomposable. A tree T with a single node V as root is the only tree decomposition of G. Since G is a simple graph, the number of edges in G can be one at most. The only vertex-induced subgraph of G with $2 \leq m \leq |V|$ vertices is itself and, thus, $|E| \leq 2m - 3 = 2|V| - 3 = 1$ holds and G is not over-constrained.

Induction hypothesis: For all $G' = (V', E')$, a subgraph of G with $|V'| < |V|$, if G' is tree decomposable, then G' is not over-constrained.

Induction step: Assume that G is tree decomposable. Then, there is a tree decomposition T of G. V is the root of T and $\{V_1, V_2, V_3\}$ is a set decomposition of G where V_i is the root of T_i, the i-th child of the root V of T, for $1 \leq i \leq 3$. Let G_i be the subgraph of G induced by V_i, for $1 \leq i \leq 3$. By induction hypothesis, G_i is not over constrained. Since a set decomposition of a graph induces a partition on the set of edges of the graph, G_i for $1 \leq i \leq 3$ are edge-disjoint graphs. This, along with induction hypothesis implies that any vertex-induced subgraph of G with m vertices, $2 \leq m \leq |V|$, must have $|E| \leq 2m - 3$ edges and, thus, G is not over constrained. □

If a graph is tree decomposable, then any subgraph obtained by removing some of its edges is also tree decomposable. This is a useful property of tree decompositions which we will make use of later on.

Theorem 4.5 *Let $G = (V, E)$ be a tree decomposable graph. For all $E' \subseteq E$, $G' = (V, E')$, the subgraph of G with the set of edges E', is tree decomposable.*

Proof

Let T be a tree decomposition of G. By Theorem 4.2, for each edge $(a, b) \in E$, there is a leaf $\{a, b\} \in T$. For any $E' \subseteq E$, T is also a tree decomposition of $G' = (V, E')$ and, thus, G' is tree decomposable. $\square$

Next, we prove an interesting result relating well constrained graphs and tree decomposable graphs. First we must prove some lemmas that show some additional properties of set decompositions of graphs.

The first lemma relates connectivity and the well constrained property of a geometric constraint graph.

Lemma 4.6 *Let $G = (V, E)$ be a connected geometric constraint graph not over constrained, if G has an articulation vertex, then G is an under constrained graph.*

Proof

Assume that a is an articulation vertex of G. Let $G_1, \ldots, G_n$ be the separation classes induced by a in G. Then define the subgraphs $G' = G_1$ and $G'' = G_2 \cup \cdots \cup G_n$. Note that $G' \cap G'' = \{a\}$. Let $G' = (V', E')$ and $G'' = (V'', E'')$. Because G is not over constrained, we know that $\text{Deficit}(G') \geq 0$ and $\text{Deficit}(G'') \geq 0$, therefore

$$2|V'| \geq 3 + |E'| \quad \text{and} \quad 2|V''| \geq 3 + |E''|$$

considering that $|V| = |V'| + |V''| - 1$, we can write

$$\text{Deficit}(G) = 2|V'| + 2|V''| - 5 - |E|$$

applying the two inequalities shown below,

$$\text{Deficit}(G) = 2|V'| + 2|V''| - 5 - |E| \geq 3 + |E'| + 3 + |E''| - 5 - |E|$$

finally, $|E| = |E'| + |E''|$ because separation classes are edge-disjoint, hence

$$\text{Deficit}(G) = 2|V'| + 2|V''| - 5 - |E| \geq 1$$

Then, because $\text{deficit}(G) \geq 1$ and G is not overconstrained, we conclude that G is under constrained. $\square$

Note that although the previous lemma requires a graph to be connected, when a graph is both not connected and not over constrained it is trivially under constrained.

The following lemma states that set decomposition of a graph preserves the property of being well constrained.

Lemma 4.7 *Let G be a well constrained geometric constraint graph and $\{V_1, V_2, V_3\}$ a set decomposition of it. Then the subgraphs induced by the set decomposition $\{G_1, G_2, G_3\}$ are well constrained graphs.*

Proof

Without lose of generality, let us choose one of the induced subgraphs, say G_1, and let us prove that G_1 is well constrained.

G_1 can be an edge or a more complex graph. If it is an edge the Lemma holds because it is a well constrained graph. If it is a graph with more than one edge, the Lemma 3.6 assures that $\{a, b\}$, the active elements of G_1, is a separation pair of G. The separating graphs induced by $\{a, b\}$ are necessarily G_1 and $G_2 \cup G_3$.

Consider $G_2 \cup G_3$, it is a not over constrained and connected graph because G is well constrained. Moreover, $G_2 \cap G_3 = \{c\}$ and c is an articulation vertex of $G_2 \cap G_3$, therefore Lemma 4.6 applies and $G_2 \cup G_3$ is under constrained.

Now, because of Lemma 3.8,

$$\text{Deficit}(G) = \text{Deficit}(G_1) + \text{Deficit}(G_2 \cup G_3) - 1$$

but G is well constrained and thus

$$\begin{gathered}\text{Deficit}(G) = \text{Deficit}(G_1) + \text{Deficit}(G_2 \cup G_3) - 1 = 0 \\ \text{Deficit}(G_1) = 1 - \text{Deficit}(G_2 \cup G_3)\end{gathered}$$

considering that $G_2 \cup G_3$ is under constrained, $\text{Deficit}(G_2 \cup G_3) \geq 1$, therefore

$$\begin{gathered}\text{Deficit}(G_1) = 1 - \text{Deficit}(G_2 \cup G_3) \\ \text{Deficit}(G_1) = 1 - \text{Deficit}(G_2 \cup G_3) \leq 1 - 1 \\ \text{Deficit}(G_1) \leq 0\end{gathered}$$

because G_1 is not over constrained $\text{Deficit}(G_1) \geq 0$, hence $\text{Deficit}(G_1) = 0$ and G_1 is well constrained. $\square$

Now the following theorem can be proved. This result shows that if a well constrained graph has a tree decomposition, then this tree decomposition is a full tree decomposition.

Theorem 4.8 *Let G be a tree decomposable geometric constraint graph and T a tree decomposition of G. G is well constrained iff T is a full tree decomposition.*

Proof
First we must prove that if G is a well constrained tree decomposable graph then it is full tree decomposable.

Assume that T is a tree decomposition of G. Because G is tree decomposable and well constrained, Lemma 4.7 applies. Then, every set decomposition of G yields well constrained graphs. Therefore, the induced graph corresponding to every node of T must be well constrained because cames from a set decomposition of a well constrained graph. Particularly, all the leaves of T induce well constrained subgraphs of G and thus must correspond to the edges of G. Consequently T is a full tree decomposition.

In the second part we must prove that a graph G for which there is a tree decomposition T with all the leaves containing an edge is a well constrained graph. This can be proved by structural induction on T. See Manna in [45] for an introduction to structural induction.

Induction base: Let T be a elementary tree decomposition with only one node $\{a, b\}$. Assume that the corresponding graph G contains one edge. Then G is trivially well constrained.

Induction hypothesis: Let T a full tree decomposition of G and T_i any child of T. Then, by induction hypothesis, the subgraph of G induced by T_i is a well constrained graph.

Induction step: Let T a full tree decomposition of $G = (E, V)$ and T_1, T_2, and T_3 their children. Let the corresponding induced subgraphs of G be $G_1 = (E_1, V_1)$, $G_2 = (E_2, V_2)$ and $G_3 = (E_3, V_3)$. Now, we should prove that G is well constrained.

Since G is full tree decomposable, Theorem 4.4 assures that G is not over constrained. Moreover, the deficit of G can be computed from the deficits of G_i,

$$\begin{aligned}
\text{Deficit}(G) &= 2|V| - 3 - |E| \\
&= 2(|V_1| + |V_2| + |V_3| - 3) - 3 - |E_1| - |E_2| - |E_3| \\
&= (2|V_1| - 3 - |E_1|) + (2|V_2| - 3 - |E_2|) + (2|V_3| - 3 - |E_3|) \\
&= \text{Deficit}(G_1) + \text{Deficit}(G_2) + \text{Deficit}(G_2)
\end{aligned}$$

But by inductive hypothesis G_i are well constrained graphs and thus their deficits are 0, therefore $\text{Deficit}(G) = 0$. A graph with deficit zero can not be under constrained, hence G is well constrained. □

A tree decomposition of the graph in Figure 4.1 is shown in Figure 4.2. Since the graph is well constrained, every leaf of the tree decomposition corresponds to an edge in the graph.

4.2 Domain of Fudos and Hoffmann's Methods

The main goal of this section is to prove that reduction analysis and decomposition analysis are two different algorithms to compute tree decompositions of a graph. While reduction analysis and decomposition analysis allow to decide whether or not a graph is solvable, tree decompositions describe the properties of solvable graphs abstractly and thus, tree decompositions are a good characterization of the domain of the constructive geometric constraint solvers.

Here we prove that the domain of reduction analysis and the domain of decomposition analysis are the same. Moreover, they can be characterized by the existence of a tree decomposition of the graph associated with the geometric constraint problem.

Lemma 4.9 *Let $G = (V, E)$ be a well constrained geometric constraint graph. Then, the following assertions are equivalent:*

1. *There is a full tree decomposition T of G.*
2. *G is solvable by reduction analysis.*
3. *G is solvable by decomposition analysis.*

Proof
In this proof we refer to the reduction systems defined in Sections 3.2 and 3.3. Therefore, the same notation is used here.

First, we prove that 1 implies 3. Assume that there is a full tree decomposition T of G. Let $V_1, \ldots, V_t$ be the list of internal nodes generated by a breath-first traversal of T. We write *Decomp*(T, V_i) to denote the set decomposition of V_i in the tree T. Since every node appears in the list after its father, the sets of nodes $\mathbf{O}_0 = \{V_1\}$ and $\mathbf{O}_i = (\mathbf{O}_{i-1} - V_i) \cup \mathit{Decomp}(T, V_i)$ are well defined. Moreover, $\mathbf{O}_{i-1} \longrightarrow_d \mathbf{O}_i$ is the reduction rule in decomposition analysis by the second condition in Definition 4.1. Since the set of leaves of T is $\mathbf{S}_G$, the set of clusters $\mathbf{O}_t$ coincides with $\mathbf{S}_G$ which implies that G is solvable by reduction analysis.

Now, we prove that 3 implies 2. Assume that $\mathbf{O}_0 = \{V\} \longrightarrow_d \cdots \longrightarrow_d \mathbf{O}_t = \mathbf{S}_G$ is a reduction sequence computed by decomposition analysis. At each reduction step $\mathbf{O}_{i-1} \longrightarrow_d \mathbf{O}_i$, for $i \in \{1, \ldots, t\}$, there are $\{C_1, C_2, C_3\} \subseteq \mathbf{O}_i$ and $C \in \mathbf{O}_{i-1}$ such that $\{C_1, C_2, C_3\}$ is a set decomposition of the subgraph of G induced by C with respect to G. Since $\{C_1, C_2, C_3\}$ is a set decomposition of C, $\mathbf{S}_j = \mathbf{O}_{t-j}$, $j \in \{0, \ldots, t\}$ is well defined and $\mathbf{S}_{i-1} \longrightarrow_r \mathbf{S}_i$ is the reduction relation in reduction analysis. Moreover, G is solvable by reduction analysis because $\mathbf{S}_G$ reduces to $\{V\}$.

Finally, we prove that 2 implies 1. Assume that there is a reduction sequence $\mathbf{S}_G = \mathbf{S}_0 \longrightarrow_r \cdots \longrightarrow_r \mathbf{S}_t = \{V\}$. For $i \in \{1, \ldots, t\}$, there are three clusters C_1, C_2, C_3 in $\mathbf{S}_{i-1}$ such that $\{C_1, C_2, C_3\}$ is a set decomposition of C and $\mathbf{S}_i = (\mathbf{S}_{i-1} - \{C_1, C_2, C_3\}) \cup C$. Moreover, every set decomposition of C is also a set decomposition of the subgraph of G induced by C because every edge of G is in a cluster of $\mathbf{S}_G$. We can assign a tree decomposition to each cluster in the sets $\mathbf{S}_i$ as follows. We assign to each cluster in $\mathbf{S}_G$ a tree with this cluster as its only node. For each reduction $\mathbf{S}_{i-1} \longrightarrow_r \mathbf{S}_i$ we assign to the new cluster C the tree decomposition with root C and sons the tree decompositions assigned to C_1, C_2, C_3. After t steps, it remains only the tree decomposition assigned to V and thus, there is a full tree decomposition of G. $\square$

4.3 Domain of Owen's Method

In the previous section we have shown that the class of full tree decomposable graphs characterizes the domain of reduction and decomposition analysis. Here, we show that the decomposition analysis studied in Section 3.4 can also be characterized by the existence of a full tree decomposition and that it has the same domain as the reduction and decomposition analysis above mentioned.

In what follows we will consider only constraint graphs G associated with well constrained problems. In these conditions, *s-trees* are binary trees whose root is G, interior nodes are modified split graphs with respect to some separation pair of the parent node and, the leaves are either triangles or triconnected graphs with no articulation pairs.

According to the number of virtual edges in the triangles in the leaf nodes of an s-tree, we classify them in four different types. See the second column in Table 4.1.

To characterize the decomposition analysis studied in Section 3.4 we prove two lemmas.

Lemma 4.10 *If a geometric constraint graph G is full tree decomposable, then G is s-tree decomposable.*

Proof

Assume that T is a full tree decomposition of G. We shall proceed by induction on the structure of T. Refer to Table 4.1.

Induction base: Let $G = (V, E)$ be a graph such that $V = \{a, b, c\}$ and $E = \{(a, b), (a, c), (b, c)\}$. The tree T in the third column of Table 4.1 is a full tree decomposition of G. Then the tree in the fourth column is a s-tree whose root is a graph $G = G_0$ with just one node representing the triangle $\{a, b, c\}$.

Induction hypothesis: Let G' be a subgraph of G. If G' is full tree decomposable then G' is s-tree decomposable.

Induction step: If $\{C_1, C_2, C_3\}$ is a set decomposition of C and $\{a, b, c\}$ the active elements, we have that $C' = C - \{a, b, c\}$, $C'_1 = C_1 - \{a, b\}$, $C'_2 = C_2 - \{a, c\}$ and $C'_3 = C_3 - \{b, c\}$.

Let G be a graph and T a full tree decomposition of G such that its root is $\{a, b, c\} \cup C'$ and the roots of its subtrees are $\{a, b\} \cup C'_1$, $\{a, c\} \cup C'_2$ and $\{b, c\} \cup C'_3$.

Assume that $C'_1 \neq \emptyset$ and $C'_2 = C'_3 = \emptyset$. This corresponds to case number 1 in Table 4.1. Let G'_1 be the subgraph induced by $\{a, b\} \cup C'_1$ in G. Let T_1 be the full tree decomposition of G'_1.

By Lemma 3.6, $\{a, b\}$ is a separation pair of G thus G'_1 is a separation graph of G. Build the other separation graph G_2 as the graph $G_2 = (V_2, E_2)$ with $V_2 = \{a, b, c\}$ and $E_2 = \{(a, c), (b, c)\}$. By Definition 3.3, G_2 is under constrained, thus by Lemma 3.9, G'_1 is well constrained.

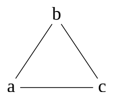
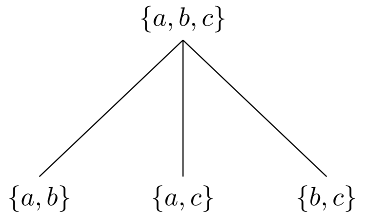
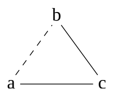
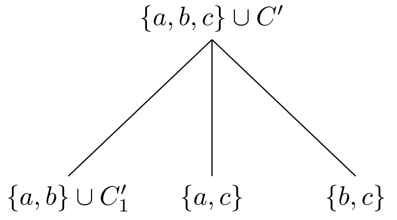
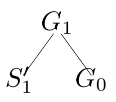
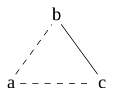
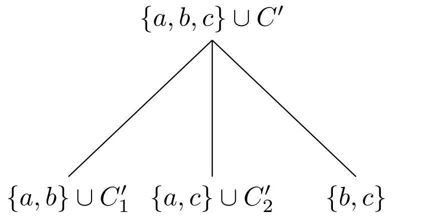
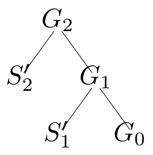
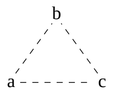
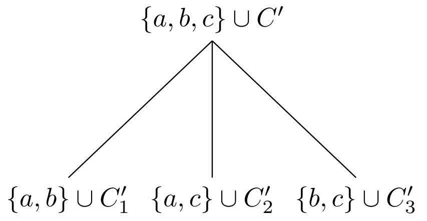
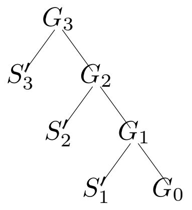
n	G_0	*Tree-decomposition* T	*S-tree* S
0			G_0
1		 $\{a,b,c\} \cup C'$ $\{a,b\} \cup C'_1$ $\{a,c\}$ $\{b,c\}$	
2		 $\{a,b,c\} \cup C'$ $\{a,b\} \cup C'_1$ $\{a,c\} \cup C'_2$ $\{b,c\}$	
3		 $\{a,b,c\} \cup C'$ $\{a,b\} \cup C'_1$ $\{a,c\} \cup C'_2$ $\{b,c\} \cup C'_3$	 G_3 S'_3 G_2 S'_2 G_1 S'_1 G_0

Table 4.1: Types of interior nodes in a tree decomposition and the equivalent s-tree decomposition.

```
**func** FromTreeToS-Tree($T$)
  $G_0$ := ComputeTriangle($T$)
  $S$ := BinaryTree($G_0$, NullTree, NullTree)
  $n$ := NumberOfVirtualEdges($G_0$)
  **for** $j$ **in** 1 **to** $n$ **do**
    $T_j$ := Subtree($T, j$)
    $S'_j$ := FromTreeToS-Tree($T_j$)
    $G'_j$ := Root($S'_j$)
    $G_j$ := MergeGraphs($G'_j, G_{j-1}$)
    $S$ := BinaryTree($G_j, S'_j, S$)
  **end**
  **return** $S$
**end**
```

Figure 4.3: Computing a s-tree S from a tree decomposition T.

Now build the modified split graphs of G as $G_0 = (V_2, E_2 \cup \{(a, b)\})$ and G'_1. Since G'_1 is full tree decomposable, by the induction hypothesis it is s-tree decomposable. Therefore there is a s-tree, say S'_1, whose root is G'_1. Hence the binary tree whose root is G and whose subtrees are G_0 and S'_1 is a s-tree. Therefore G is s-tree decomposable.

Applying the same procedure for cases $C'_2 \neq \emptyset$ and $C'_3 \neq \emptyset$, which correspond to cases 2 and 3 in Table 4.1, completes the proof. $\square$

If function `ComputeTriangle(`T`)` computes the triangle associated with a node of a tree decomposition, and function `MergeGraphs(`G_1, G_2`)` rebuilds a graph from its modified split graphs, Figure 4.3 shows an algorithm that, based on Lemma 4.10, computes a s-tree S from a tree decomposition T of a graph G.

Notice that, as built in Table 4.1 for well constrained problems, s-trees exhibit the following nice property. The splits in the s-tree S that generate n virtual edges, $1 \leq n \leq 3$, in any triangle leaf node G_0 take place in the immediate n predecessors of G_0. We say that these s-trees are *regular*. If there is a regular s-tree corresponding to a geometric constraint graph G, we say that G is *regular s-tree decomposable.*

Lemma 4.11 *Let G be a geometric constraint graph. If G is regular s-tree decomposable, then G is full tree decomposable.*

Proof

Assume that S is a regular s-tree whose root is G. Again we shall proceed by induction on the structure of S. Refer to Table 4.1.

Induction base: Let $G = (V, E)$ be a graph such that $V = \{a, b, c\}$ and $E = \{(a, b), (a, c), (b, c)\}$. The regular s-tree S of G is that given in the fourth column of Table 4.1. Then the tree given in the third column is a full tree decomposition of G.

Induction hypothesis: Let G' be a subgraph of G. If G' is regular s-tree decomposable then G' is full tree decomposable.

Induction step: Let S be a regular s-tree whose root is G and whose subtrees are G_0 and S'_1.

Assume that $G_0 = (V_0, E_0)$ with $V_0 = \{a, b, c\}$ and $E_0 = \{(a, c), (b, c)\} \cup \{(a, b)\}$ and let $G'_1 = (V'_1, E'_1)$ be the root of the regular s-tree S'_1. By Definition 3.12, G_0 and G'_1 are the modified split graphs of G with respect to the separation pair $\{a, b\}$. Since G'_1 is regular s-tree decomposable, by induction hypothesis it is full tree decomposable. Therefore there is a full tree decomposition T'_1 of G'_1.

Build a full tree decomposition T such that its subtrees are T'_1, $\{a, c\}$ and $\{b, c\}$. See Table 4.1. Subtrees $\{a, c\}$ and $\{b, c\}$ share the node c. Since $\{a, b\}$ is a separation pair of G, $V_0 \cap V'_1 = \{a, b\}$. But $c \in V_0$, thus $c \notin V'_1$, $\{a, c\} \cap V'_1 = \{a\}$, and $\{b, c\} \cap V'_1 = \{b\}$. Therefore T is a full tree decomposition of G.

Applying the same procedure for cases in rows three and four in Table 4.1 completes the proof. $\square$

But s-trees are not necessarily regular. We will show that considering regular s-trees does not entail any loss of generality.

Lemma 4.12 *Let G be a geometric constraint graph. If G is s-tree decomposable, then G is regular s-tree decomposable.*

Proof

To prove this Lemma we supply the algorithm in Figure 4.4 that outputs a regular s-tree. This is a variation of the algorithm in Figure 3.14.

Algorithm in Figure 3.14 contains the following sentence:

$$G_1, G_2 := \text{SeparatingGraphs}(G)$$

Note that a graph can have more than two separating graphs, therefore this sentence has an undeterministic meaning. By properly choosing the separating graphs in each step of the s-tree construction we can assure that a regular s-tree is built. This leads to an actual instance of the original algorithm in which undeterminism has been used to favor the construction of regular s-trees.

The new version of the algorithm is written according to the cases shown in the Table 4.1. We classify the graph to be decomposed into one of the four cases depicted in the second column of the Table. Finally we build the resulting s-tree for every kind of graph following the pattern shown in the fourth column of the Table. The result is a regular s-tree.

The problem is now to classify the graph into one of the four kinds. This is done using the set of sepating pairs of the graph. Figure 4.4 shows the modified algorithm.

□

4.3.1 Transforming a general s-tree to a regular one

In this section we give a procedure to transform a non regular s-tree into a regular one. Notice that this procedure is a different proof of Lemma 4.12.

Figure 4.5 shows an s-tree which is not regular because of virtual edges (b, e) and (e, f) in the leaf nodes labeled G_0 and S_1 respectively. From now on we will say that edges (b, e) and (e, f) in Figure 4.5 are not regular. The subtree denoted S_3 is irrelevant and is not fully expanded. Let us first define the concept of *equivalent* s-tree decompositions as follows

Definition 4.13 *Let S and S' be two different s-tree decompositions of a well-constrained graph G. We say that S and S' define equivalent decompostions of G if they have the same set of leaf nodes. We will also say that S and S' are equivalent.*

```
func RegularAnalysis($G$)
  L := SeparatingPairs($G$)
  if $\exists a, b, c : \{a, b\} \in L \wedge \{b, c\} \in L \wedge \{a, c\} \in L$ then /* case 3 */
    $G_1$,$G_2$ := SeparatingGraphsByPair($G$,a,b)
    AddVirtualEdge($G_1$,$G_2$)
    Assume $G_2 \ni \{a, c\}$
    $G_{21}$,$G_{22}$ := SeparatingGraphsByPair($G_2$,a,c)
    AddVirtualEdge($G_{21}$,$G_{22}$)
    Assume $G_{22} \ni \{b, c\}$
    $G_{221}$,$G_{222}$ := SeparatingGraphsByPair($G_{22}$,b,c)
    AddVirtualEdge($G_{221}$,$G_{222}$)
    $S$ := BinaryTree($G$, Analysis($G_1$),
              BinaryTree($G_2$, Analysis($G_{21}$),
                BinaryTree($G_{22}$, Analysis($G_{222}$), Analysis($G_{222}$)))
  elseif $\exists a, b, c : \{a, b\} \in L \wedge \{a, c\} \in L$ then /* case 2 */
    $G_1$,$G_2$ := SeparatingGraphsByPair($G$,a,b)
    AddVirtualEdge($G_1$,$G_2$)
    Assume $G_2 \ni \{a, c\}$
    $G_{21}$,$G_{22}$ := SeparatingGraphsByPair($G_2$,a,c)
    AddVirtualEdge($G_{21}$,$G_{22}$)
    $S$ := BinaryTree($G$, Analysis($G_1$),
              BinaryTree($G_2$, Analysis($G_{21}$),Analysis($G_{22}$))
  elseif $\exists a, b : \{a, b\} \in L$ then /* case 1 */
    $G_1$,$G_2$ := SeparatingGraphsByPair($G$,a,b)
    AddVirtualEdge($G_1$,$G_2$)
    $S$ := BinaryTree($G$, Analysis($G_1$), Analysis($G_2$))
  elseif $L = \emptyset$ then /* case 0 */
    $S$ := BinaryTree($G$, nullTree, nullTree)
  fi
  return $S$
end
```

Figure 4.4: Analysis algorithm modified to compute regular s-trees.

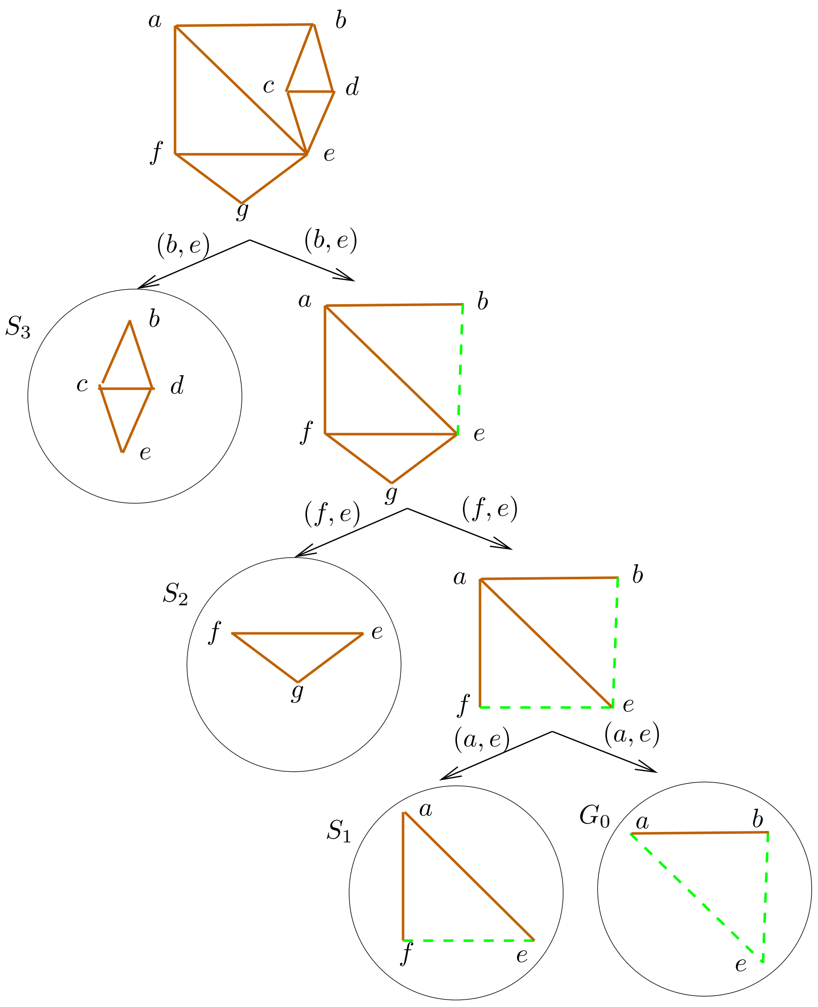


Figure 4.5: Graph and s-tree. Subtree S_3 is not fully expanded.

The goal now is to prove the following lemma.

Lemma 4.14 *Let S be an s-tree computed from a well-constrained graph G. Let (a, b) be a virtual edge which is not regular in S. Then S always can be transformed into an s-tree, S', such that S and S' are equivalent and that the virtual edge (a, b) is regular in S'.*

We prove this claim by proving the correctness of the algorithm given in Figure 4.6, from now on `EdgeRegularization()` algorithm. Figure 4.7 illustrates how the algorithm works. On the left, Figure 4.7 depicts a subtree of an abstract s-tree S whose root is the graph G_n and whose leaf node G_0 includes the virtual edge (a, b), introduced by the split of G_n. We denote this subtree by $\mathcal{S}$.

Figure 4.7 right shows the subtree in the transformed s-tree S' yielded by the algorithm `EdgeRegularization()`. We denote this subtree by $\mathcal{S}'$. If S is an s-tree and G is a graph, then the graph given by `MergeGraphs(Root(`S`),`G`)` will be denoted $S \uplus G$.

We begin proving the following lemma.

Lemma 4.15 *Trees $\mathcal{S}$ and $\mathcal{S}'$ have the same set of leaf nodes. Moreover, their root nodes represent the same graph.*

Proof

The set of leaf nodes in both the s-tree $\mathcal{S}$ and the tree $\mathcal{S}'$ is given by the union of the leaf nodes of the subtrees S_i, $1 \leq i \leq n$, plus the leaf node G_0.

The root G'_n of $\mathcal{S}'$ is

$$\begin{aligned}
G'_n &= S_{n-1} \uplus G'_{n-1} \\
&= S_{n-1} \uplus (S_{n-2} \uplus G'_{n-1}) \\
&\dots \\
&= S_{n-1} \uplus (S_{n-2} \uplus \dots \uplus (S_1 \uplus (S_n \uplus G_0)) \dots) \\
&= G_n
\end{aligned}$$

□

Now we prove that the tree $\mathcal{S}'$ is an s-tree where the virtual edge (a, b) is regular. The proof is in two steps. First we prove that the smallest subtree in $\mathcal{S}'$ where (a, b) is included, is an s-tree and (a, b) is regular in it.

```
func EdgeRegularization ()
  INPUT
    $S$ : s-tree decompostion of a graph $G$.
    $G_0$ : non regular leaf node in $S$ corresponding to a triangle.
    $\{a, b\}$ : separation pair that defines a non regular virtual
      edge, $(a, b)$, in $G_0$.
  OUTPUT
    $S'$ : s-tree that defines equivalent decompositions of $G$ as $S$ and
      where the virtual edge $(a, b)$ is regular.

  1. Compute the node $G_n$ in $S$ where the separation pair $\{a, b\}$
     introduced the virtual edge $(a, b)$ which is not regular in $G_0$.
  2. In the tree whose root is $G_n$, identify the subtree $S_n$ whose split
     graph includes the actual edge $(a, b)$.
  3. Build the s-tree $S'_1$ whose root is the graph
     $G'_1 = \text{MergeGraphs}(\text{Root}(S_n), G_0)$ and whose
     subtrees are $S_n$ and the s-tree of the graph $G_0$.
  4. for $i$ in 2 to n do
       Build the s-tree $S'_i$ whose root is the graph
       $G'_i = \text{MergeGraphs}(\text{Root}(S_{i-1}), G'_i)$ and whose
       subtrees are $S_{i-1}$ and $S'_{i-1}$.
     end
  5. Replace in $S$ the subtree whose root is $G_n$ with the tree built in step 4.
  6. return $S$
end
```

Figure 4.6: Virtual edge regularization.

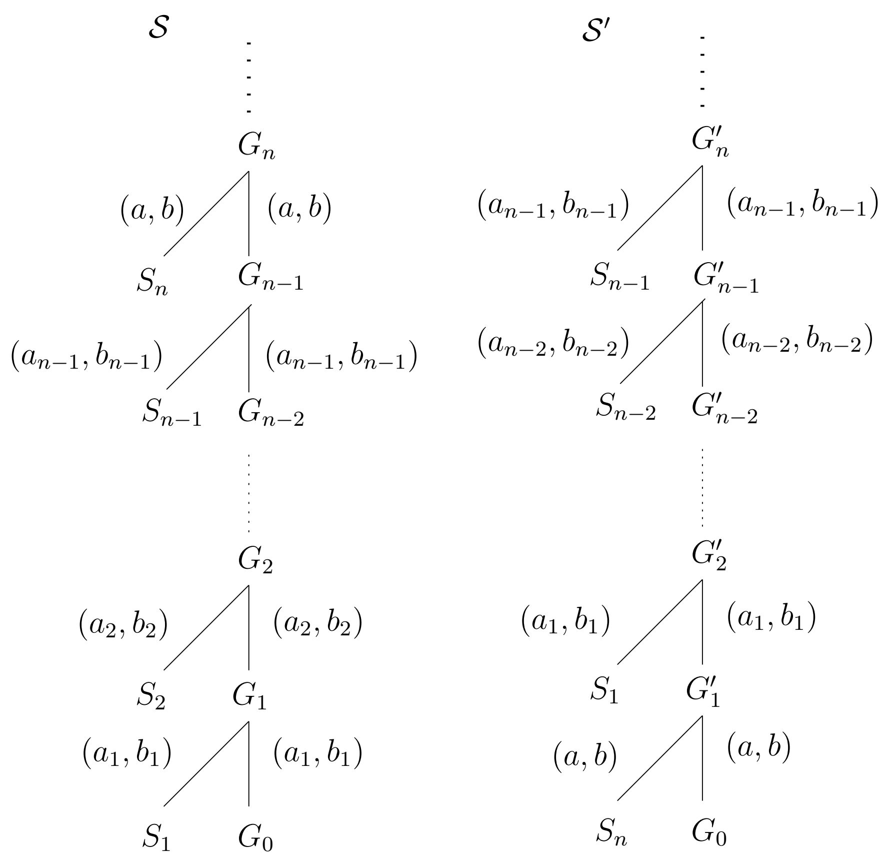


Figure 4.7: Left: Initial subtree. Virtual edge (a, b) in G_0 is not regular. Right: Modified subtree. Virtual edge (a, b) in G_0 is regular.

Lemma 4.16 *The subtree S'_1 in $\mathcal{S}'$ whose root is the graph $G'_1 = S_n \uplus G_0$, and whose subtrees are S_n and G_0, is an s-tree. The virtual edge (a, b) is regular in S'_1.*

Proof
S_n is a subtree of the s-tree $\mathcal{S}$. Therefore S_n is an s-tree. By definition (see Table 4.1) G_0 is an s-tree. S_n and G_0 are subtrees of $\mathcal{S}$ that only have in common the pair $\{a, b\}$ which separates $S_n \uplus G_0$ into Root(S_n) and G_0. Therefore S'_1 is an s-tree.

The virtual edge (a, b) generated by the separation pair $\{a, b\}$ is included in S_n and, by construction, the virtual edge (a, b) is not included in $S_n \uplus G_0$. Therefore the virtual edge (a, b) in G_0 is regular. $\square$

Next we prove that the tree generated by the algorithm is actually an s-tree.

Lemma 4.17 *The tree $\mathcal{S}'$ is an s-tree.*

Proof
We need to prove that the split of each subtree root in $\mathcal{S}$ is defined by a separation pair.

By construction, the separation pair (a, b) that splits G'_1 in $\mathcal{S}'$ is the separation pair that splits G_n in $\mathcal{S}$.

For the remaining separation pairs, let i denote an arbitrari separation pair in $\mathcal{S}$ with $1 \leq i \leq n$.

By construction $G'_i = S_{i-1} \uplus G'_{i-1}$. But $G'_{i-1} = S_n \uplus G_{i-2}$. Thus $G'_i = S_{i-1} \uplus (S_n \uplus G_{i-2})$. In the s-tree $\mathcal{S}$, S_{i-1} and G_{i-2} are separated by $\{a_{i-1}, b_{i-1}\}$. Moreover, S_{i-1} and S_n share at most the separation pair $\{a, b\}$ when $a = a_{i-1}$ and $b = b_{i-1}$. See Figure 4.8. Therefore, $\{a_{i-1}, b_{i-1}\}$ separates $S_{i-1}) \uplus (S_n \uplus G_{i-2})$ into S_{i-1} and $S_n \uplus G_{i-2} = G'_{i-1}$. $\square$

Figure 4.9 shows the s-tree output by the algorithm `EdgeRegularization()` when the input is the s-tree given in Figure 4.5 and the virtual edge to be regularized is (b, e), in the leaf node G_0. The subtree S_3 is not fully expanded. Notice that the virtual edge (e, f) in subtree S_1 still is not regular. Repeatedly applying the algorithm to the s-tree output by the previous iteration, yields an s-tree where all the virtual edges are regular. Therefore we have the following result

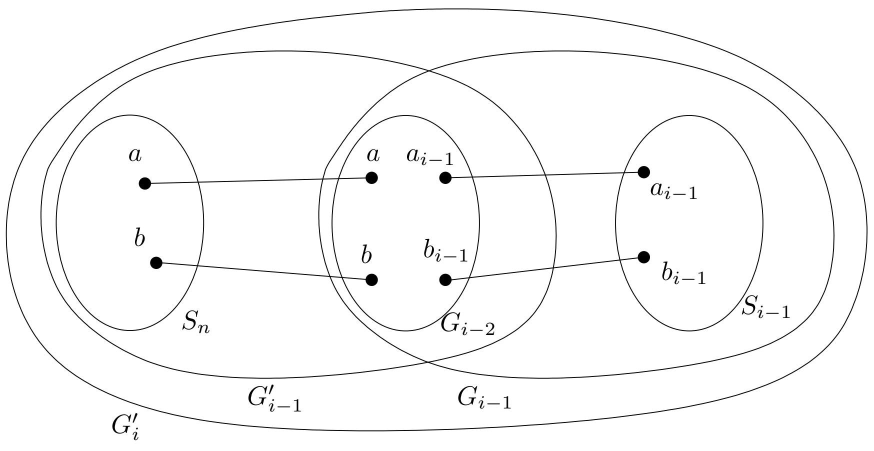


Figure 4.8: Separation pair common to S_n and S_{i-1} subtrees.

Corollary 4.18 *If G is s-tree decomposable then we can compute a regular s-tree decomposition of G.*

4.4 Domain Equivalence of Constructive Methods

Now, we see that the class of s-tree decomposable geometric constraint graphs and the class of tree decomposable graphs are the same. In other words, a geometric constraint problem expressed by means of a geometric constraint graph is solvable by Owen's technique if and only if the graph is full tree decomposable. We proved in Lemma 4.9 that a geometric constraint graph is solvable by reduction or decomposition analysis of Fudos and Hoffmann, [20, 21], if and only if the graph is full tree decomposable. Therefore Owen's technique, decomposition and reduction analysis have the same domain and this domain can be characterized by the class of full tree decomposable graphs.

Theorem 4.19 *Let $G = (V, E)$ be a geometric constraint graph. The following assertions are equivalent:*

1. *G is full tree decomposable.*
2. *G is s-tree decomposable.*

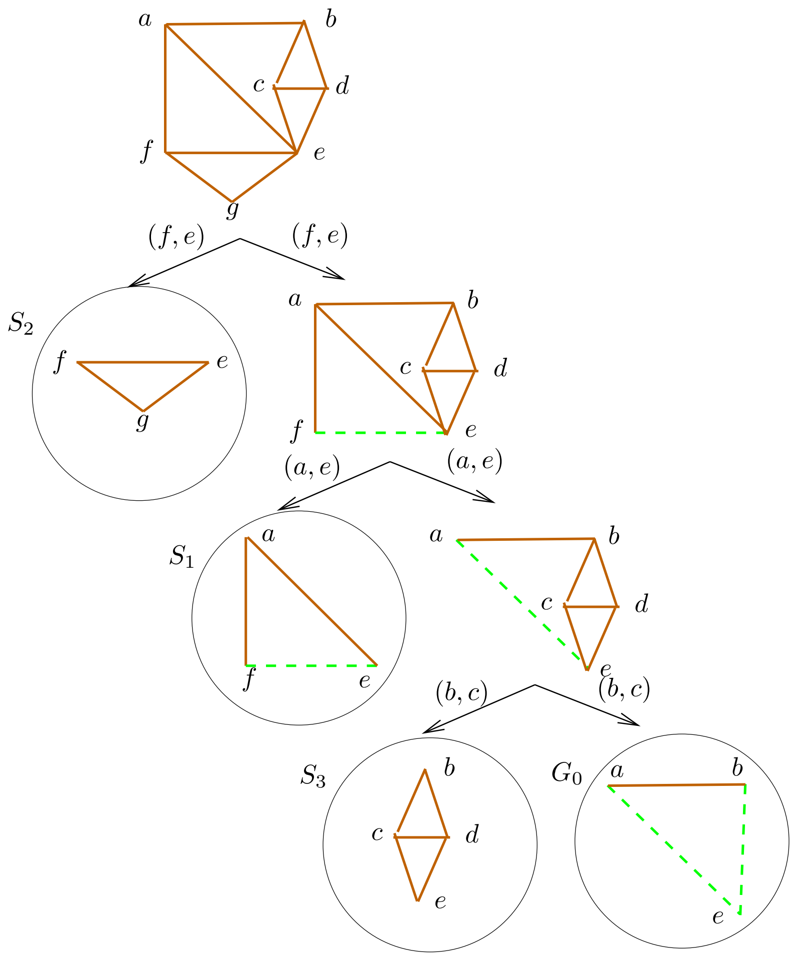


Figure 4.9: Regular s-tree derived form the s-tree in Figure 4.5. Subtree S_3 is not fully developed.

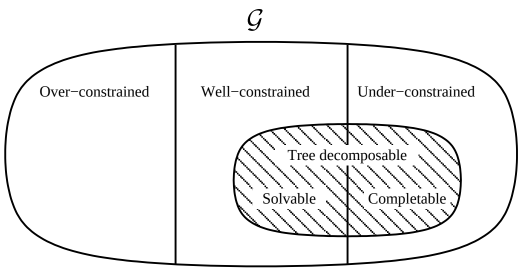


Figure 4.10: Classification of geometric constraint graphs according to Definition 3.3 and the set of tree decomposable graphs.

3. G is solvable by reduction analysis.

4. G is solvable by decomposition analysis.

Proof

Lemma 4.9 proves the equivalence of assertions 1, 3 and 4. Lemma 4.10 proves that 1 implies 2. Finally, Lemmas 4.11 and 4.12 prove that 2 implies 1. □

Theorem 4.19 shows that the constructive methods considered here have the same domain, that is, they solve the same class of problems. However, each method exhibits different properties like efficiency and behavior with respect to under constrained problems, see [21, 50].

We define this distinguished class of graphs as follows.

Definition 4.20 *Let G be a geometric constraint graph. We say that G is* solvable *iff G is full tree decomposable.*

Assume that $\mathcal{G}$ denotes the set of geometric constraint graphs. The definitions of over, well and under constrained graphs induce a partition on $\mathcal{G}$, see Figure 4.10. By Theorem 4.4, tree decomposable graphs are a subset of the graphs which are not over constrained. The class of tree decomposable graphs overlaps the set of well constrained graphs and the set of under constrained graphs. By Theorem 4.8, the solvable graphs (or full tree decomposable graphs) are those well constrained graphs which are tree decomposable.

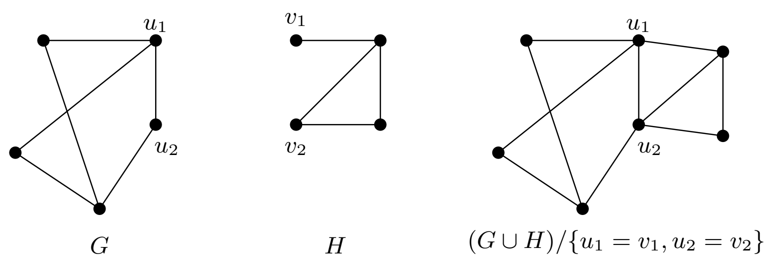


Figure 4.11: Two vertex amalgamation $(G \cup H)/\{u_1 = v_1, u_2 = v_2\}$.

In Chapter 5, we will consider the class of under constrained graphs which are tree decomposable. We call the graphs in this class *completable* graphs.

The proof of Theorem 4.19 is constructive in the sense that algorithms to compute tree decompositions can be derived from it. These algorithms compute a tree decomposition in polynomial worst case time on the number of vertices of the graph. This time bound is justified because tree decompositions are computed in polynomial time from the output of the algorithms described in [20, 21, 50], which are polynomial.

4.5 Generation of Solvable Graphs

In this Section we present a method to generate the class of solvable graphs. To describe the method we begin by recalling the vertex amalgamation of graphs, see Gross and Yellen [24].

Definition 4.21 *Let G and H be disjoint graphs with $u_1, u_2 \in V(G)$ and $v_1, v_2 \in V(H)$. The* two vertex amalgamation $(G \cup H)/\{u_1 = v_1, u_2 = v_2\}$ *is the graph obtained from the union $G \cup H$ by merging the vertex u_1 of G and v_1 of H in a single vertex named u_1 and merging the vertex u_2 of G and v_2 of H in a single vertex named u_2.*

Figure 4.11 shows an example of graph amalgamation.

Definition 4.22 *Let G and H two disjoint graphs with $u, v \in V(G)$ and $(e, f) \in E(H)$. We define the* substitution of (e, f) in H by G modulo $\{u, v\}$ *as the new graph obtained by deleting the edge (e, f) from H and*

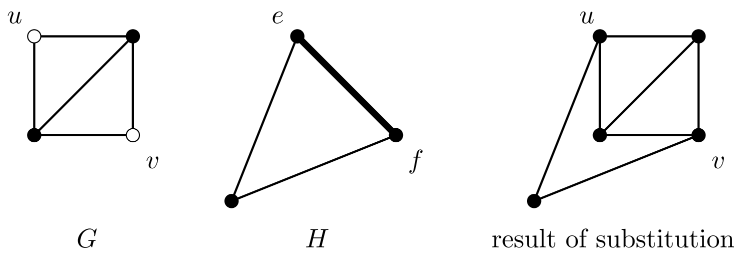


Figure 4.12: Substitution of (e, f) in H by G modulo $\{u, v\}$.

amalgamating this with G. That is

$$\big(G \cup (H - (e, f))\big)/\{u = e, v = f\}$$

The meaning of this definition is illustrated in Figure 4.12. In this figure, the edge substituted in graph H is drawn in thick line and the relevant vertices of graph G are drawn as hollow circles.

Now, using the substitution operation, we define a method to compute the class of solvable graphs. We first define a class of graphs $\mathcal{T}$ and then we prove that this class is equivalent to the class of solvable geometric constraint graphs. The class $\mathcal{T}$ is defined as follows.

Definition 4.23 *We define the set of graphs $\mathcal{T}$ by the following rules:*

1. *The triangle graph belongs to $\mathcal{T}$.*
2. *Let $G, H \in \mathcal{T}$ be arbitrary graphs of the class not necessarily distinct. Choose an arbitrary edge (e, f) from H and an arbitrary pair of vertices $\{u, v\}$ from G. Then the graph obtained by substituting (e, f) in H by G modulo $\{u, v\}$ is also a graph of $\mathcal{T}$.*
3. *No other element belongs to $\mathcal{T}$.*

The previous definition has a constructive nature. Therefore, it is easy to build elements belonging to the class $\mathcal{T}$ applying it. Figure 4.13 depicts an example of some elements of $\mathcal{T}$ and their constructive definition.

The next step is to prove that the class $\mathcal{T}$ is the class of solvable graphs. We first prove the two following lemmas.

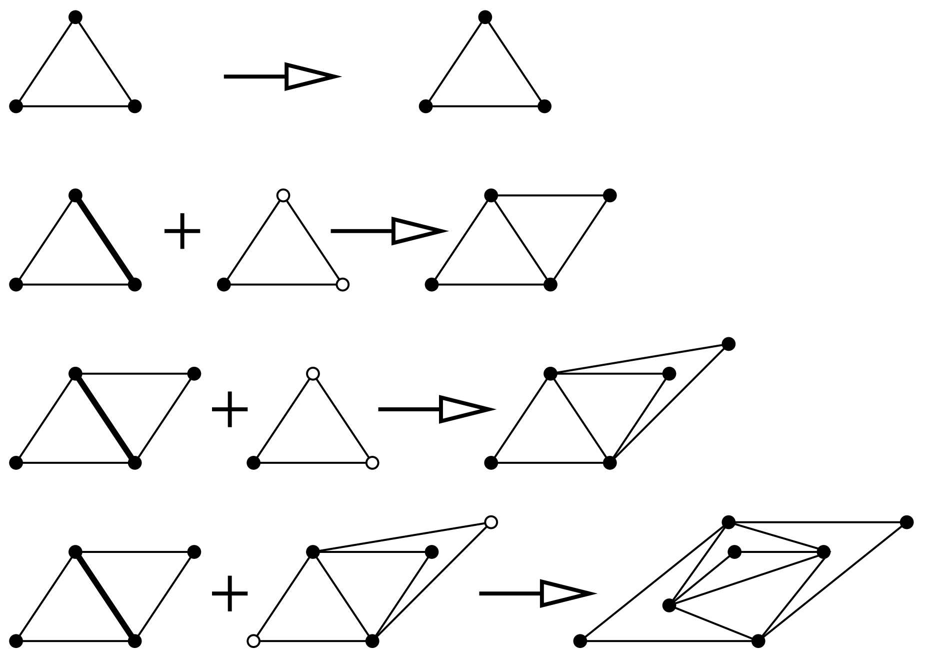


Figure 4.13: Some elements of $\mathcal{T}$

Lemma 4.24 *If G is a graph that belongs to $\mathcal{T}$, then G is solvable.*

Proof

We prove that every element of $\mathcal{T}$ is full tree decomposable by structural induction on the set $\mathcal{T}$.

Induction base: Let $T \in \mathcal{T}$ be a triangle. Then T is trivially full tree decomposable.

Induction hypothesis: Let $G' \in \mathcal{T}$ be a graph obtained from the substitution of edge (e, f) in $H \in \mathcal{T}$ by $G \in \mathcal{T}$ modulo $\{u, v\}$. Assume that G and H are full tree decomposable graphs.

Induction step: Let us prove that G' is a full tree decomposable graph. Refer to Figure 4.14.

Assume that T_G and T_H are tree decompositions of G and H respectively. Then, since (e, f) is an edge of H, T_H must have a leaf node with exactly two vertices $\{e, f\}$. Let $P = \langle p_1, \ldots, p_n = \{e, f\}\rangle$ be the path from the root to the leaf $\{e, f\}$ in T_H. Now we build a new tree $T_{G'}$ from T_H by the following procedure:

1. Replace every ocurrence of e in the tree T_H by u.
2. Replace every ocurrence of f in the tree T_H by v.
3. To every node in the path P, add the vertices of $V(G) \setminus \{u, v\}$.
4. Replace the leaf p_n in P by the tree T_G.

The new tree $T_{G'}$ is a full tree decomposition of G' and hence G' is full tree decomposable. □

Lemma 4.25 *Let $G = (V, E)$ be a geometric constraint graph with $|V| \geq 3$. If G is solvable, then G belongs to $\mathcal{T}$.*

Proof

We prove that every full tree decomposable graph G with more than 2 vertices belongs to $\mathcal{T}$. Given that G is full tree decomposable, there exists a full tree decomposition T of G. We shall prove the result by using induction on the structure of T.

Induction base: Let G be a graph with $V(G) = 3$ which is full tree decomposable. Then, the graph is a triangle and thus it belongs to $\mathcal{T}$ by definition.

Induction hypothesis: Let G be a graph and T its full tree decomposition. Let the children of T be $\{T_1, T_2, T_3\}$ and $\{G_1, G_2.G_3\}$ the corresponding subgraphs induced on G. We assume that, $\forall i \in \{1, \ldots, 3\}$, if $V(G_i) \geq 3$ then G_i belongs to $\mathcal{T}$.

Induction step: Let us prove that, under induction hypothesis, G belongs to $\mathcal{T}$. Consider the set decomposition of T, $\{T_1, T_2, T_3\}$. Notice that at least one element in $\{T_1, T_2, T_3\}$ must contain more than two elements, otherwise T is a triangle. According to whether $\{T_1, T_2, T_3\}$ has one, two or three members with more than two geometric elements, we distinguish three cases.

Case 1: Let $T_1 = \{a, b\}$, $T_2 = \{b, c\}$ and $T_3 = \{a, c, d, \ldots\}$, refer to Figure 4.15. Thus, the induced subgraphs G_1 and G_2 contain one edge. Moreover, by induction hypothesis, G_3 the induced subgraph of T_3 belongs to $\mathcal{T}$. Therefore G can be defined by substitution from elements of $\mathcal{T}$ by the following procedure. First consider the triangle graph H with vertices $\{a', b, c'\}$ and the corresponding edges. H belongs to $\mathcal{T}$ by definition. After that,

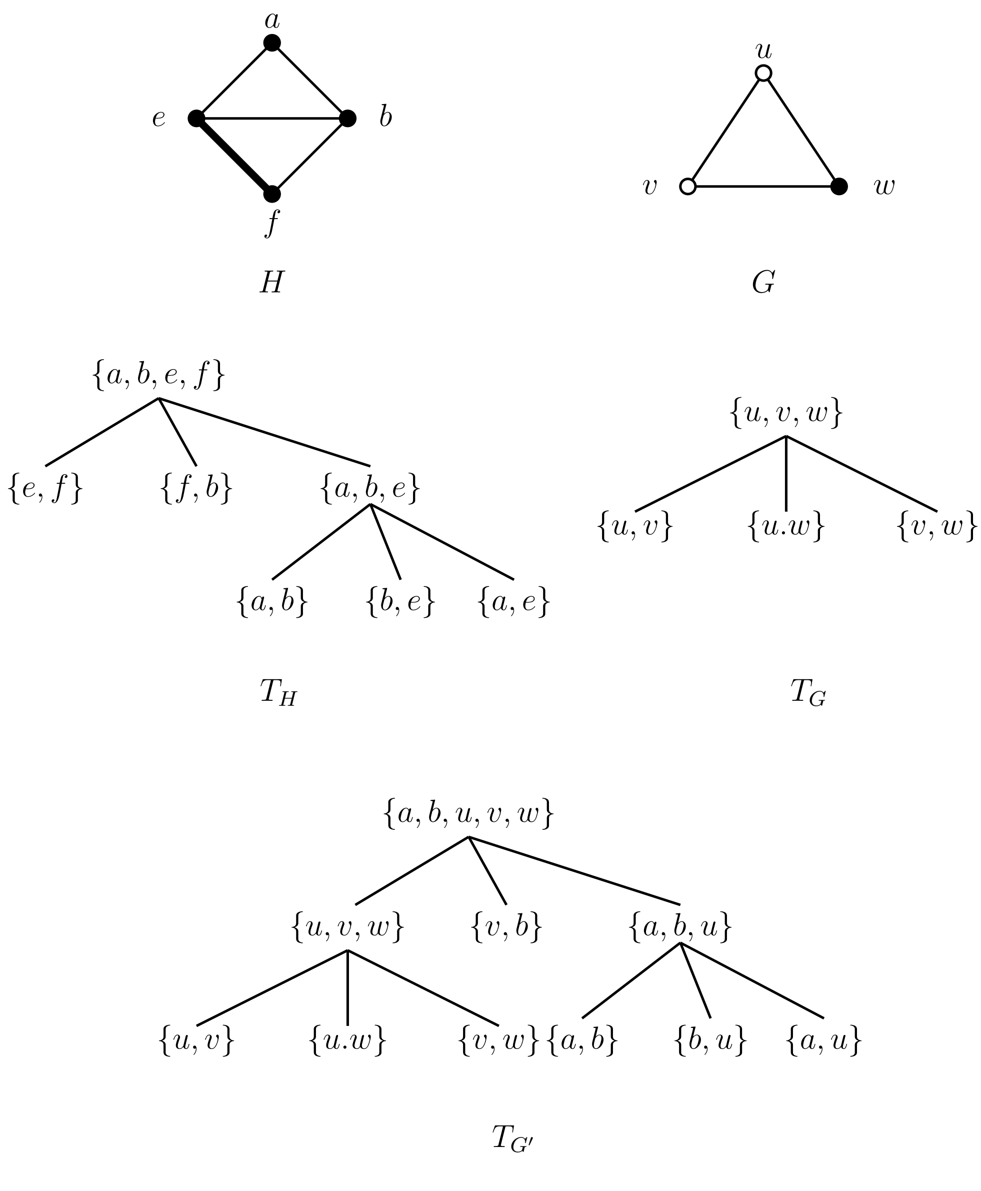


Figure 4.14: Construction of a tree decomposition for G' from G and H.

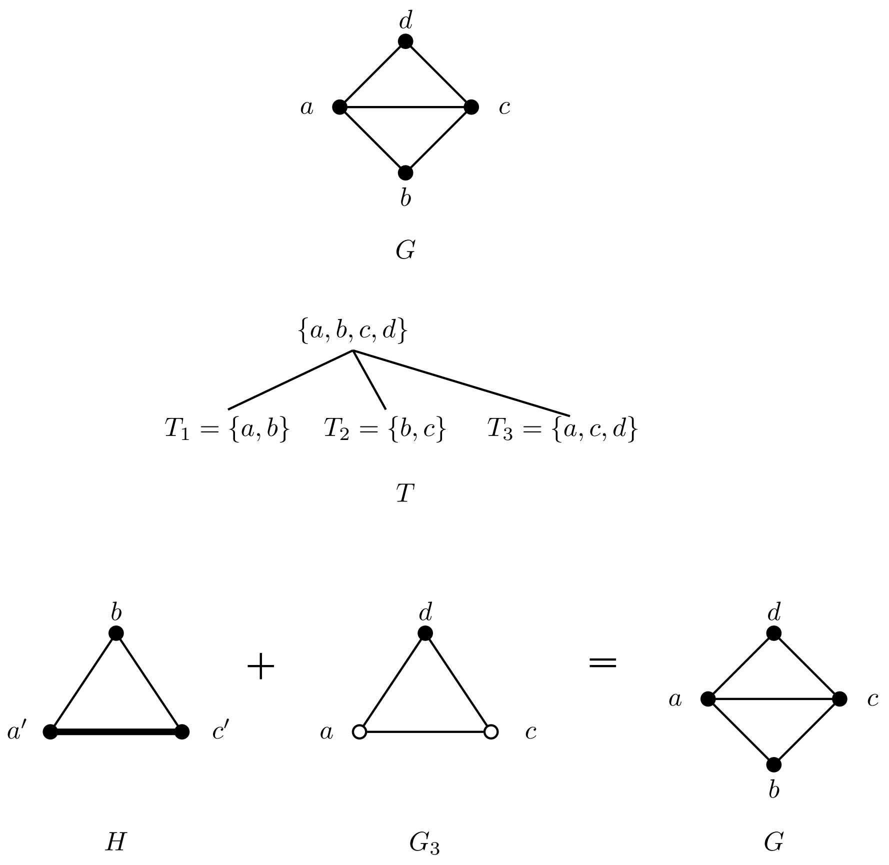


Figure 4.15: Construction of graph G as a member of $\mathcal{T}$.

substitute edge (a', c') of H by G_3 modulo $\{a, c\}$. The new graph is exactly G and, by construction, it belongs to $\mathcal{T}$.

Cases two and three are proved by applying the same technique. □

Theorem 4.26 *The class $\mathcal{T}$ and the class of solvable constraint graphs are the same.*

Proof
Apply Lemma 4.24 and Lemma 4.25. □

4.6 Summary

In this chapter we have defined the class of tree decomposable geometric constraint graphs by extending the concept of set decomposition of a graph. Some interesting properties of this class have been proved. We have also defined the class of full tree decomposable graphs.

Then, we have shown that the domain of the Fudos and Hoffmann's and Owen's analysis methods is the class of full tree decomposable graphs. This class has been named the class of solvable graphs.

Finally, we have given a method to generate the class of solvable graphs. This method recursively defines the class as sequence similar to Henneberg sequences. We have proved that the class of graphs generated by this method is the class of full tree decomposable graphs. The generation schema trivially shows that the class of solvable graphs is a well founded set and thus structural induction can be used to prove properties in it.

5 The Completion Operation

In this chapter we investigate the problem of transforming an under constrained geometric constraint graph into a well constrained one by adding additional constraints. We refer to this transformation as *completion.* We distinguish two kinds of completions: free completion and conditional completion.

Both free and conditional completions should be considered as new operations added to the architecture given in Chapter 2. In Chapter 6, these operations will allow us to solve the problem of synchronizing different views in a concurrent engineering environment.

Most of the work in this chapter has been previously published in [36, 40, 41].

5.1 The Graph Completion Problem

Existing geometric constraint solving techniques have been developed under the assumption that problems are well constrained, that is, that the number of constraints and their placement on the geometric elements define a problem with a finite number of solutions for non-degenerate configurations.

There are a number of scenarios where the assumption of well constrained does not apply. An example is the early stages of the design process when only a few parameters are established. The problem then is under constrained, that is, it has an infinite number of solutions for non degenerate configurations.

Another situation arises in cooperative design systems. Here, different activities in product design and manufacture examine different subsets of the information in the object's model. The presentation of such an information subset has been called a *view*, [13]. Maintaining consistent views is a central issue. To solve this problem, Hoffmann and Joan-Arinyo introduced in [26] the *constraint schema reconciliation* concept. It is assumed that with each view there is associated a geometric constraint graph. If the graphs are different, they are reconciled by drawing constraints from one of the graphs and adjoining them to the other graph. However, how to actually select the adjoined constraints is not defined.

To automate product design and manufacture, techniques to solve under constrained problems and the schema reconciliation problem should be devised. If the geometric constraint problem is represented by a geometric constraint graph, solving under constrained problems and the schema reconciliation problem can be seen as specific instances of the *geometric constraint graph completion* problem or just the *completion* problem. This problem is defined as follows:

Problem 5.1 (Well constrained completion) *Given the geometric constraint graph associated to an under constrained geometric constraint problem, add automatically new edges (constraints) to the graph in such way that the corresponding geometric constraint problem is well constrained.*

Although not much attention has been paid to the well constrained completion problem, its solution is not obvious at all. Results from distinct fields, such as combinatorial rigidity theory and matroid theory, must be taken into account to achieve an optimal solution for this problem. In Section 5.6 we briefly review these results and we discuss how to apply them to the solution of the well constrained completion problem.

Only complete geometric constraint techniques solve any well constrained geometric constraint graph, see [27]. However, the generality of complete solvers prevents them from always finding a symbolic construction plan. Then, numerical techniques must be applied to compute the solution. Constructive geometric constraint solvers, like those in [20, 21, 50], are incomplete. However, they always compute a symbolic construction plan if the analyzed geometric constraint graph is in the domain of the solver. In this chapter we study techniques to complete geometric constraint graphs. The result must be a graph that can be solved by constructive techniques. This problem, which is the object of this chapter, is defined as follows:

Problem 5.2 (Completion) *Given the geometric constraint graph associated to an under constrained geometric constraint problem, add automatically new edges (constraints) to the graph in such way that the corresponding geometric constraint problem is solvable.*

We report here on a technique that solves Problem 5.2 in two different scenarios. In one, the extra constraints are automatically defined without any further condition. In the other one, the extra constraints are drawn from an independently given set of constraints defined on the geometries of the problem at hand. In the first scenario, the technique always generates a solvable problem if the initial problem is *completable*, i.e., all subproblems can be solved by a constructive technique.

In the second scenario, it is not always possible to generate a solvable graph, at least in a first step. This situation arises, for instance, when there are not enough extra constraints. However, the algorithm always computes a completable graph, a graph that can be completed by applying the first technique.

5.2 Completability of Under Constrained Graphs

Consider an under constrained tree decomposable geometric constraint graph. To effectively solve it, first we should transform the graph into a solvable one. This amounts to adding new constraints to the graph, that is, adding new edges to the constraint graph.

If $G = (V, E)$ is the geometric constraint graph, Definition 3.3 gives the number of constraints that must be added to G to transform it into a well constrained graph. However, deciding which constraints should actually be added to the graph is not an straightforward matter, because it could result in either an over constrained graph or a well constrained graph that is not solvable.

We will use the term *completion* to refer to the process of adding new constraints (edges) to an under constrained graph. The resulting graph will be named the *completed* graph or just *a completion*.

Figure 5.1 left shows an under constrained graph. Notice that there is just one edge incident to node e and that node g has no incident edges. The

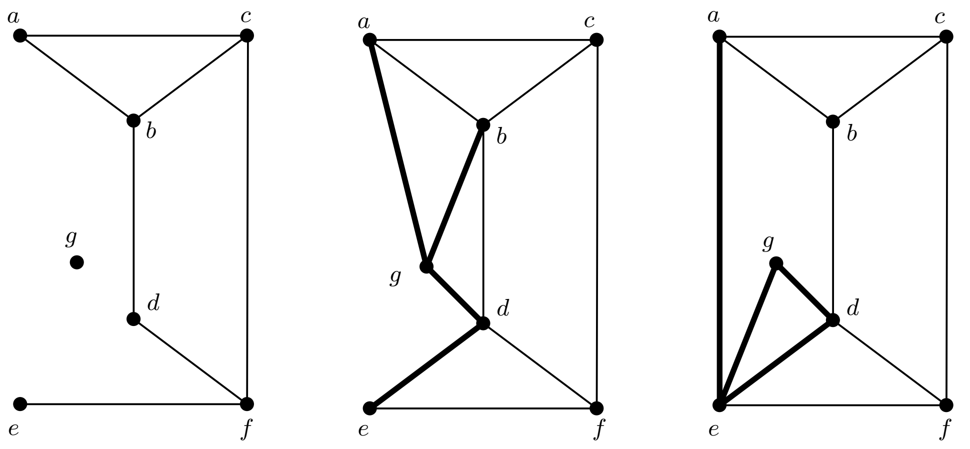


Figure 5.1: A graph and two different possible completions.

completion shown in Figure 5.1 middle is a solvable and thus well constrained graph. Moreover, the completion shown in Figure 5.1 right is an unsolvable graph although it is well constrained.

We are interested in completed graphs that are solvable.

Definition 5.1 *Let $G = (V, E)$ be an under constrained geometric constraint graph. We say that G is* completable *if there is a solvable graph $G' = (V, E \cup E')$ which is a completion of G.*

In the following sections we present two techniques to complete completable graphs: free completion and conditional completion.

5.3 Free Completion

Among the different ways to complete an under constrained graph G one can think of, we are interested first in one where the only additional requirement is that edges added to G must be taken from the set

$$E_F = \{(a, b) | a, b \in V \text{ and } (a, b) \notin E\}$$

We will call this a *free completion* of graph G.

```
Algorithm FreeCompletion
1. Compute a tree decomposition $T$ of $G$.
2. Compute the set of edges
       $E' = \{(a,b) \mid \{a,b\} \text{ is a leaf of } T \text{ and } (a,b) \notin E\}$
3. Complete the graph $G$ as $G' = (V, E \cup E')$.
```

Figure 5.2: Free completion algorithm for graph $G = (V, E)$.

The free completion problem has been previously addressed by Fudos and Hoffmann in [21]. There, a new method was developed to deal specifically with under constrained problems. Our approach is based on tree decompositions and, since tree decompositions do not depend on any geometric constraint solving technique, is a general method.

The free completion algorithm is based on the following result.

Proposition 5.2 *Let $G = (V, E)$ be an under constrained geometric constraint graph. G is completable if and only if there is a tree decomposition of G.*

Proof

Assume that $G = (V, E)$ is a completable graph. Then, there is a solvable graph $G' = (V, E \cup E')$. A solvable graph has a full tree decomposition. Let T be a full tree decomposition of G'. Since $E \subseteq E \cup E'$, by Theorem 4.5, G is tree decomposable.

Now assume that G is tree decomposable. Let T be a tree decomposition of G. Let E' be the set of leaves of T that do not correspond to an edge in E,

$$E' = \{(a,b) \mid \{a,b\} \text{ is a leaf of } T \text{ and } (a,b) \notin E\}.$$

The graph $G' = (V, E \cup E')$ is a completion of G and T is a full tree decomposition of G'. Thus, G is completable. $\square$

The algorithm to carry out the free completion is given in Figure 5.2.

Consider the constraint graph given in Figure 5.1 left and assume that step 1 in algorithm `FreeCompletion` computes the tree decomposition T shown in

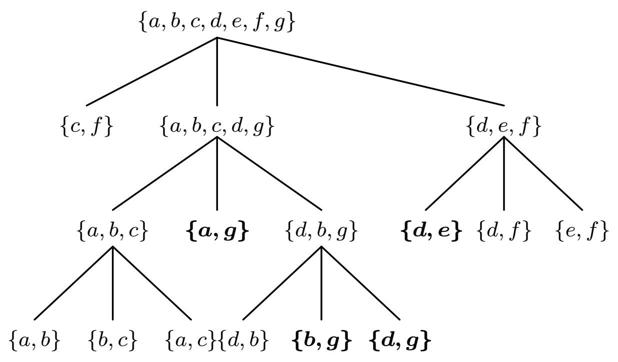


Figure 5.3: Tree decomposition of the graph in Figure 5.1 left.

Figure 5.3. The leaves of T whose nodes do not define an edge in the graph depicted in Figure 5.1 left are (a,g), (d,e), (b,g) and (d,g). Therefore, the set of extra edges computed in step 2 is

$$E' = \{(a,g),(d,e),(b,g),(d,g)\}.$$

Figure 5.1 in the middle shows the constraint graph yielded by the algorithm `FreeCompletion`.

Notice that, in general, a tree decomposition T of a constraint graph $G = (V,E)$ is not unique. Therefore, a free completion of a graph G is not necessarily unique.

5.4 Conditional Completion

Conditional completion is the second technique to complete geometric constraint graphs presented in this chapter. The study of conditional completion was originally motivated by the constraint schema reconciliation problem, [26]. Chapter 6 discusses this problem and how conditional completion allows to solve it.

Let $G = (V,E)$ be a completable geometric constraint graph. Let $\widehat{G} = (V,\widehat{E})$ be a geometric constraint graph whose edges $\widehat{E}$ define a set of additional constraints between geometric elements in V with $\widehat{E} \cap E = \emptyset$. Notice that if $\widehat{E} \cap E \neq \emptyset$, we always can consider as additional edges those in the set $\widehat{E}' = \widehat{E} \setminus E$.

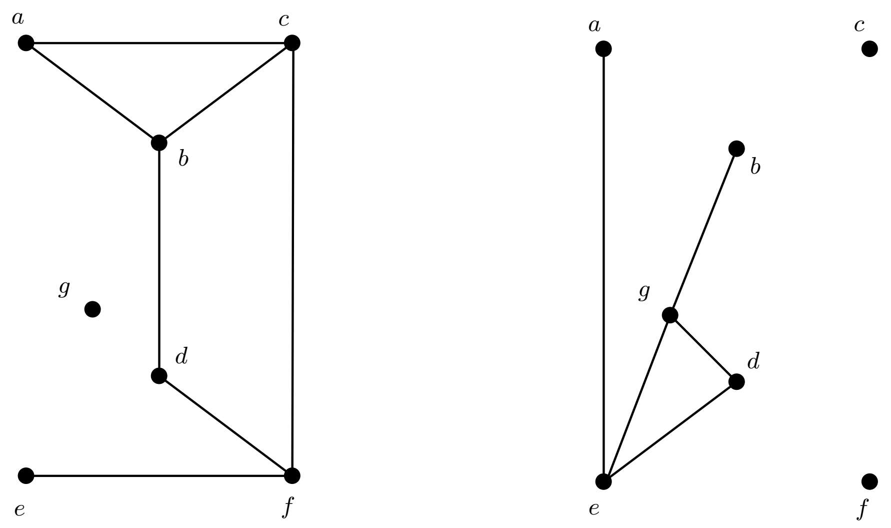


Figure 5.4: Geometric constraint graphs G (left) and $\widehat{G}$ (right).

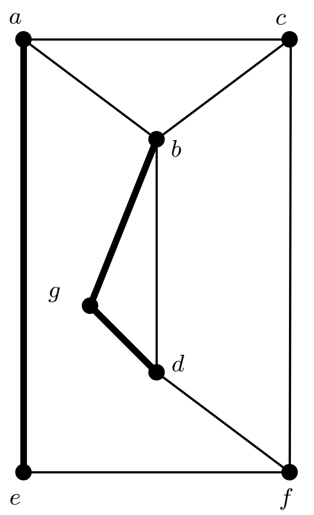
Our interest is to build a solvable completion of G by adding edges drawn from $\widehat{E}$. We will refer to this completion as *conditional completion* of graph G from $\widehat{G}$. Formally, we define a conditional completion as follows.

Definition 5.3 *Let $G = (V, E)$ be a completable geometric constraint graph and let $\widehat{G} = (V, \widehat{E})$ be a geometric constraint graph. $G' = (V, E')$ is a* conditional completion *of G from $\widehat{G}$ if G' is a completable completion of G and $E' = E \cup \widehat{E}'$ with $\widehat{E}' \neq \emptyset$ and $\widehat{E}' \subseteq \widehat{E}$.*

We call the graph $\widehat{G}$ the *additional graph.* Figure 5.4 shows an under constrained graph G (left) and an additional graph $\widehat{G}$ (right). Figure 5.5 shows two conditional completions of G from $\widehat{G}$. The graph on the left is completable and the graph on the right is solvable.

Among all the possible conditional completions of an under constrained graph with respect to a given additional graph, we are interested in those that are maximal in the following sense.

Definition 5.4 *Let $G' = (V, E')$ be a conditional completion of G from $\widehat{G}$. We say that G' is a* maximal conditional completion *if for any other*



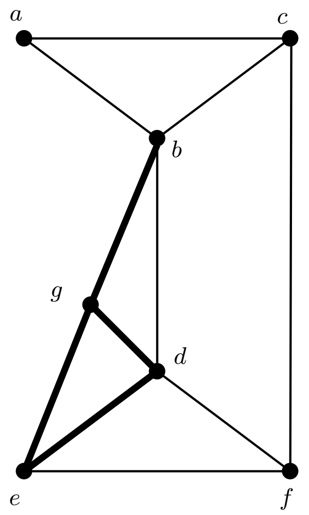


Figure 5.5: Conditional completion (left) and maximal conditional completion (right) of G from $\widehat{G}$ in Figure 5.4.

conditional completion $G'' = (V, E'')$ *from* $\widehat{G}$*, the relation* $|E''| \leq |E'|$ *holds.*

A maximal conditional completion does not need to result in a solvable graph. Notice that it is not possible to build a solvable completion if the number of edges in $\widehat{E}$ is smaller than the number of edges required by Definition 3.3 to transform G into a well constrained graph. In these conditions, however, a solvable graph still can be built applying free completion to the graph yielded by the maximal conditional completion.

5.4.1 Maximal Conditional Completion as a Combinatorial Optimization Problem

Following Papadimitriou *et al.* [54], a *subset system* $S = (\mathcal{E}, \mathcal{S})$ is a finite set $\mathcal{E}$ together with a collection $\mathcal{S}$ of subsets of $\mathcal{E}$ closed under inclusion (that is, if $A \in \mathcal{S}$ and $A' \subseteq A$, then $A' \in \mathcal{S}$). The elements of $\mathcal{S}$ are called *independent*. The *combinatorial optimization problem associated with* a subset system $(\mathcal{E}, \mathcal{S})$ is the following: Given a *weight* $w(e) \geq 0$ for each $e \in \mathcal{E}$, find an independent subset that has the largest possible total weight.

Let $G = (V, E)$ be an under constrained graph and let $\widehat{G} = (V, \widehat{E})$ be the additional graph. Assume that E and $\widehat{E}$ are disjoint, $E \cap \widehat{E} = \emptyset$. If we define $\mathcal{E} = E \cup \widehat{E}$ and $\mathcal{S}$ as the subsets of $\mathcal{E}$ such that are tree decomposable, then S is a subset system. We say that a set of edges $E' \subseteq \mathcal{E}$ is tree decomposable if the graph $G' = (V, E')$ is tree decomposable. Notice that by Theorem 4.5, $\mathcal{S}$ is closed under inclusion.

We define the weight function $w(e)$ over the set of edges $\mathcal{E} = E \cup \widehat{E}$ as

$$w(e) = \begin{cases} 1 & \text{if } e \in \widehat{E} \\ 2 & \text{if } e \in E \end{cases}$$

In these conditions, the problem of computing a maximal conditional completion of G from $\widehat{G}$ is a combinatorial optimization problem associated with the subset system S and the weight function w.

5.4.2 The Greedy Algorithm

The greedy algorithm is the simplest algorithm to solve a combinatorial optimization problem associated with a subset system. In this section we first describe the greedy algorithm for subset systems. Then we apply it to the maximal conditional completion problem.

Figure 5.6 shows the greedy algorithm on the subset system $S = (\mathcal{E}, \mathcal{S})$. The algorithm computes I, an independent set in S which we want to have the largest total weight. We recall from Papadimitriou, [54], that it is not necessary to explicitly compute S to answer the question $I \cup \{e\} \in \mathcal{S}$ in line 2.3 of the algorithm. It suffices to have an algorithm to decide whether the set $I \cup \{e\}$ is in S.

Since we have formalized the maximal conditional completion of a graph G from an additional graph $\widehat{G}$ as a combinatorial optimization problem, we can use the greedy algorithm in Figure 5.6 as is, without any change. Notice that, for the maximal conditional completion problem, the elements of $\mathcal{S}$ are the subsets of $\mathcal{E}$ such that they are tree decomposable. Therefore, deciding whether $I \cup \{e\}$ is in $\mathcal{S}$ is equivalent to decide whether there is a tree decomposition of the set of edges $I \cup \{e\}$.

Unfortunately, the greedy algorithm does not solve the maximal conditional completion problem. Let us show a counterexample.

Let G be the completable graph in Figure 5.4 left and let $\widehat{G}$ be the geometric

```
Algorithm GreedyOptimize
  1.    $I := \emptyset$
  2.    while $\mathcal{E} \neq \emptyset$ do
  2.1       $e :=$ a maximum weight member of $\mathcal{E}$
  2.2       $\mathcal{E} := \mathcal{E} \setminus \{e\}$
  2.3       if $I \cup \{e\} \in \mathcal{S}$ then $I := I \cup \{e\}$
```

Figure 5.6: The greedy algorithm for subset systems.

constraint graph in Figure 5.4 right. We define the subset system S and the weight function w as in Section 5.4.1.

The greedy algorithm in Figure 5.6 first selects the edges in G because their weights are larger than the weights of edges in $\widehat{G}$. Hence

$$I = \{(a,b),(a,c),(b,c),(b,d),(c,f),(d,f),(e,f)\}$$

After that, the edges in $\widehat{G}$ are considered. Because all the edges in $\widehat{G}$ have the same weight, they can be chosen at random. Assume that the sequence of edges chosen is (a,e), (b,g) and (d,g). Then, no more edges can be added to I and the independent set returned by the greedy algorithm is

$$I \quad = \quad \{(a,b),(a,c),(b,c),(b,d),(c,f),(d,f),(e,f),(a,e),(b,g),(d,g)\}$$

See Figure 5.5 left.

Now assume that the sequence of chosen edges is (e,d), (e,g), (d,g) and (g,b). At this point, no more edges can be added to I and the independent set returned by the greedy algorithm is

$$I \quad = \quad \{(a,b),(a,c),(b,c),(b,d),(c,f),(d,f),(e,f),(e,d),(e,g),(d,g),(g,b)\}$$

See Figure 5.5 right. Clearly, this independent set includes more edges than the one computed before, that is, it has larger total weight. Therefore, the greedy algorithm does not necessarily yields a maximal conditional completion.

5.5 Experimental Study of the Greedy Algorithm for the Conditional Completion Problem

The greedy algorithm does not always solve the maximal conditional completion problem. Therefore, it makes sense to investigate how far the completions yielded by the greedy algorithm are from being maximal.

Let $G = (V, E)$ be an under constrained graph and let $\widehat{G} = (V, \widehat{E})$ be the additional graph from where G must be completed. Let $\mathcal{G}(G, \widehat{G})$ denote the number of extra edges actually added to G from $\widehat{G}$ by the greedy algorithm in an specific run. Recall that since extra edges with the same weight are selected at random, the greedy algorithm is not deterministic. Let $\mathcal{O}(G, \widehat{G})$ denote the number of extra edges taken from $\widehat{G}$ that results in a maximal completion of G.

The goal of the experimental study is to measure how different $\mathcal{G}(G, \widehat{G})$ and $\mathcal{O}(G, \widehat{G})$ are for a large enough set of pairs $(G, \widehat{G})$. But computing $\mathcal{O}(G, \widehat{G})$ entails having an algorithm that solves the maximal completion problem. Therefore, we compute an estimation for $\mathcal{O}(G, \widehat{G})$.

Two different tests were conducted depending on the strategy used to define the additional set of edges.

5.5.1 First test

Test Definition

The pairs of graphs $G(V, E)$ and $\widehat{G} = (V, \widehat{E})$, with $|V| = n$, have been defined as follows:

1. A constructively solvable graph $F = (V, E_F)$ with $|V| = n$, is randomly generated. This is done by building a random tree decomposition T of $G(V, E)$ with $E = \emptyset$. Then E_F is defined as

$$E_F = \{(a, b) | a, b \in V \text{ and } (a, b) \text{ is a leaf of } T\}$$

2. The set E_F is randomly split into three pairwise disjoint subsets E, D, and O such that $E_F = E \cup D \cup O$.

3. Let K_n be the complete graph with n vertices. Let A be a set of edges randomly drawn from $E(K_n) \setminus E$. Then $\widehat{E}$ is defined as $\widehat{E} = O \cup A$.

Notice that by construction of G and $\widehat{G}$, all the edges in $O \subseteq \widehat{E}$ can be added to G. Therefore the following inequality holds:

$$0 \leq |O| \leq \mathcal{O}(G, \widehat{G}) \leq 2|V| - 3 - |E|$$

Thus, if $\mathcal{G}(G, \widehat{G}) < |O|$ then $\mathcal{G}(G, \widehat{G}) < \mathcal{O}(G, \widehat{G})$. This means that the completion yielded by the greedy algorithm is not maximal.

Data Description

A series of sets of graphs each containing twenty pairs $(G = (V, E), \widehat{G} = (V, \widehat{E}))$ where generated. Values of $|V| = n$ were $\{5, 6, \ldots, 100\}$. The greedy algorithm was serially fed with each pair. For each run, the values of $|E|$, $|\widehat{E}|$, $|O|$, and the number of edges actually added to G, $\mathcal{G}(G, \widehat{G})$, were recorded.

Results Analysis

The ratio of runs for which the completion was not maximal, that is, the number of edges actually added to G, $\mathcal{G}(G, \widehat{G})$, was smaller than $|O|$, with respect to the total number of runs was 3.4%.

The difference between $\mathcal{G}(G, \widehat{G})$, and the number of edges that could be added, $|O|$, was measured by the ratio of missing edges

$$\frac{|O| - \mathcal{G}(G, \widehat{G})}{|O|}$$

Figure 5.7 shows the number of runs versus the ratio of missing edges. For example, there were 13 runs where the ratio was 2%, 10 runs with a ratio of 3% and so on.

Finally the ratio of cases where the resulting completion is well constrained, $|E \cup \widehat{E}| = 2|V| - 3$, with respect to the total number of runs was 7.8%.

5.5.2 Second Test

Test Definition

The pairs of graphs $(G(V, E), \widehat{G}(V, \widehat{E}))$ are defined as follows:

1. A constructively solvable graph $F = (V, E_F)$ is built as described for the first test.

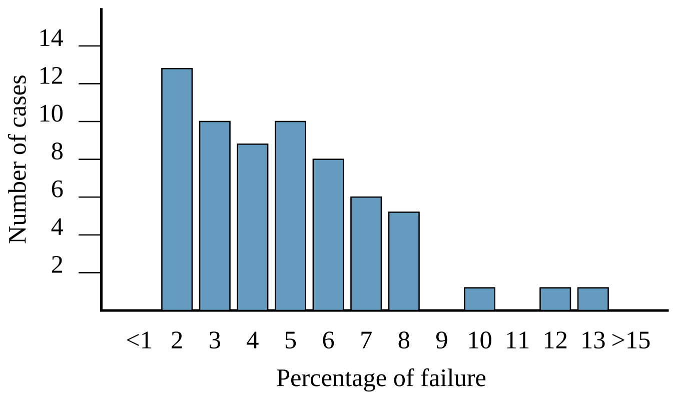


Figure 5.7: Distribution of runs where more edges in $\widehat{G}$ could be added to G.

2. The set of edges E_F is split into two subsets E and D with $E \cap D = \emptyset$ and $E_F = E \cup D$.

3. If K_n is the complete graph with n vertices, the additional set of edges is now $\widehat{E} = E(K_n) \setminus E$.

Notice that since all the edges needed to transform the under constrained graph G into a well constrained one are in $\widehat{E}$, the number of edges that can be added is given by $\mathcal{O}(G, \widehat{G}) = 2|V| - 3 - |E|$. Therefore, for this test the following inequality holds:

$$0 \leq \mathcal{G}(G, \widehat{G}) \leq \mathcal{O}(G, \widehat{G}) = 2|V| - 3 - |E|$$

Data Description

A series of sets of graphs each containing ten pairs $(G = (V, E), \widehat{G} = (V, \widehat{E}))$ where generated. Values of $|V| = n$ were $\{5, 10, 15, \dots, 100\}$. The data recorded for each run was the same as in the first test, $|E|$, $|\widehat{E}|$, $|O|$, and $\mathcal{G}(G, \widehat{G})$.

Results Analysis

In this test we counted the number of runs $(G, \widehat{G})$ for which $\mathcal{G}(G, \widehat{G}) < \mathcal{O}(G, \widehat{G})$. That is the number of runs where the greedy algorithm did not yield a well constrained completion. There were exactly 3 cases, that is 1.5% of the runs.

5.5.3 Experimental Study Conclusions

From the experimental study we can drawn the following results:

1. The greedy algorithm does not compute the maximal conditional completion in a small fraction of cases. This fraction can be estimated in about 3.5%.
2. When the greedy algorithm does not find a maximal conditional completion, the ratio of missing edges needed to generate a maximal completion with respect to the total number of edges in the well constrained graph, was in most of the runs less than 8%.
3. If the additional set of edges includes as many edges as needed to generate a well constrained completion, only in 1.5% of the runs the completion yielded was under constrained.

Therefore, the greedy algorithm is of practical application to compute conditional completions.

5.6 Well Constrained Conditional Completion

So far, we have focused our interest in completing a geometric constraint graph in such a way that the completed graph was solvable by a constructive technique. Here we relax this requirement and we only expect the completion to be well constrained; that is, we address Problem 5.1.

We proceed as in Section 5.4. First, we define the well constrained conditional completion. The main difference with respect to Definition 5.3 is that here the geometric constraint graph is required to be just under constrained instead of completable.

Definition 5.5 *Let $G = (V, E)$ be an under constrained geometric constraint graph and let $\widehat{G} = (V, \widehat{E})$ be a geometric constraint graph. $G' = (V, E')$ is a* well constrained conditional completion *of G from $\widehat{G}$ if G' is a well constrained completion of G and $E' = E \cup \widehat{E}'$ with $\widehat{E}' \neq \emptyset$ and $\widehat{E}' \subseteq \widehat{E}$.*

Next, we define the maximal well constrained conditional completion. Notice the analogy with Definition 5.4.

Definition 5.6 *Let $G' = (V, E')$ be a well constrained conditional completion of G from $\widehat{G}$. We say that G' is a* maximal well constrained conditional completion *if for any other well constrained conditional completion $G'' = (V, E'')$ from $\widehat{G}$, the relation $|E''| \leq |E'|$ holds.*

Now, we reformulate the maximal well constrained conditional completion problem as a combinatorial optimization problem associated with a subset system. Let $G = (V, E)$ be an under constrained graph and let $\widehat{G} = (V, \widehat{E})$ be the additional graph. Assume that E and $\widehat{E}$ are disjoint, $E \cap \widehat{E} = \emptyset$. If we define $\mathcal{E} = E \cup \widehat{E}$ and $\mathcal{S}$ as the subsets E' of $\mathcal{E}$ for which the graph $G' = (V(E'), E')$ is not over constrained, then S is a subset system. We define the weight function $w(e)$ over the set of edges $\mathcal{E}$ as in Section 5.4.1.

Two well established results are needed to show that the greedy algorithm always solves the maximal well constrained conditional completion problem. These results come from matroid theory and combinatorial rigidity theory, respectively.

Matroids are subset systems which fulfill some additional properties. See [53] for an in depth study of matroid theory. A well known result in combinatorial optimization is that the greedy algorithm solves any combinatorial optimization problem associated with a subset system which is a matroid, [54].

Geometric constraint graphs whose vertices are points in the plane and whose edges are distance constraints has been extensively studied in the context of combinatorial rigidity theory. The result we are interested in is the following, [23].

Theorem 5.7 *Let $G = (V, E)$ be a geometric constraint graph whose vertices are points in the plane and whose edges are distance constraints. Let $\mathcal{E} = E$ and let $\mathcal{S}$ denote the collection of subsets of $\mathcal{E}$ such that they are not over constrained. Then the subset system $S = (\mathcal{E}, \mathcal{S})$ is a matroid.*

From this, we can conclude that the greedy algorithm solves the maximal well constrained conditional completion for graphs whose vertices are points in the plane and whose edges are distance constraints. This result has interesting practical implications. For example, any complete geometric constraint solving technique can benefit from the use of the greedy algorithm to solve the maximal well constrained conditional completion problem. Moreover, it can also solve over constrained problems by applying the technique described in Section 5.7.3.

5.7 Applications of the Conditional Completion

We present three applications of the maximal conditional completion problem: constraint schema reconciliation, constraints with priorities and solving over constrained problems. We also include a case study consisting in an over constrained problem in which we require the topological constraints to be kept in the final solution.

5.7.1 Constraint Schema Reconciliation

In cooperative design systems, different activities in product design and manufacture aim at different subsets of the information in the model under design. Each of those subsets of information has been called a *view*, [13].

In this context, maintaining views consistently is a central issue. Chapter 6 is devoted to this problem and offers a solution founded on the conditional completion.

5.7.2 Constraints with priorities

In constraint-based design there are situations where it would be useful to associate a different level of priority to different constraints. For example when there is a set of mandatory constraints but the designer still has the freedom to fix the remaining constraints. We can generalize the maximal conditional completion problem and include constraints with levels of priority.

Let M be the maximum level of priority. Let E_i, $1 \leq i \leq M$, be a partition of the set of edges $\mathcal{E}$. Then, we can define a collection of graphs $G_i = (V, E_i)$, $1 \leq i \leq M$, on the same set of vertices V where V contains the endpoints of the edges in $\mathcal{E}$. Let $\mathcal{S}$ be the set of subsets of $\mathcal{E}$ which are tree decomposable.

Then $S = (\mathcal{E}, \mathcal{S})$ is a subset system. If for each $e \in \mathcal{E}$, we define the weight function $w(e) = i$ if $e \in E_i$, we get a combinatorial optimization problem on the subset system S.

Then, the greedy algorithm in Figure 5.6 applies. If $G_M = (V, E_M)$ is completable, the constraints in E_M are guaranteed to be in the resulting graph. The edges on the remaining sets E_i, for $1 \leq i < M$, are successively included in the resulting graph according to the levels of priority.

5.7.3 Over constrained problems

We first state what we understand by solving an over constrained problem.

Problem 5.3 *Let $G^* = (V, E^*)$ be an over constrained geometric constraint graph associated to an over constrained problem. The geometric constraint graph $G = (V, E)$ is a* solution of the over constrained problem *if E is a subset of E^* with the largest possible cardinality such that G is completable.*

Let $G^* = (V, E^*)$ be an over constrained graph. We define the graphs G and $\widehat{G}$ as follows.

$$\begin{aligned} G &= (V, \emptyset) \\ \widehat{G} &= (V, E^*) \end{aligned}$$

Now, we apply the greedy algorithm to compute a maximal conditional completion of G from $\widehat{G}$. If the resulting graph is completable, a free completion can be applied to get a solvable graph. Recall that an over constrained graph has a subgraph with more edges than needed, but it can have subgraphs with less edges than needed. Therefore, we can not get, in general, a solvable graph just by selecting a subset of edges in E^*.

Notice that due to the non-deterministic nature of the greedy algorithm each run could result in a different completable (or solvable) graph.

5.7.4 Case study

To illustrate how the ideas presented here work, consider the over constrained problem given by an end user and shown in Figure 5.8. The geometric elements in the problem include four points and four straight segments.

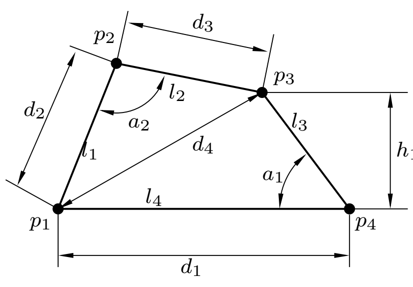


Figure 5.8: Over constrained sketch

The constraints are four point-point distances, two angles and one point-segment distance.

Assume that the over constrained graph $G^* = (V, E^*)$ associated with the sketch is the one given in Figure 5.9. And assume that, with the aim of keeping the sketch topology, the following priorities are defined on the constraints

$$w(e) = \begin{cases} 1 & \text{if } e \in E^* \text{ is a dimensional constraint} \\ 2 & \text{if } e \in E^* \text{ is a topological constraint} \end{cases}$$

If graphs G and $\widehat{G}$ are defined as in Section 5.7.3, Figure 5.10 shows a constraint graph output by the greedy algorithm. Recall that the greedy algorithm is not deterministic and that the actual output depends on the specific run.

Now, a different annotation on the input sketch, shown in Figure 5.11, which is constructively solvable, can be suggested to the end user.

5.8 Summary

In this chapter we have addressed two techniques to solve the completion problem: free completion and maximal conditional completion. The completion of a geometric constraint graph consists in adding extra edges to an under constrained graph. In free completion, the extra constraints are automatically defined without any further condition. In maximal conditional completion, the extra edges are drawn from the set of edges of a second graph which we call the additional graph. The resulting graph in both cases

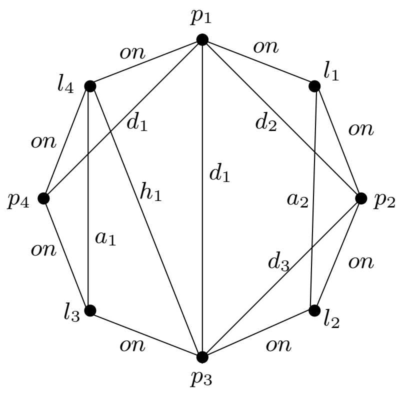


Figure 5.9: Over constrained graph associated with the problem in Figure 5.8.

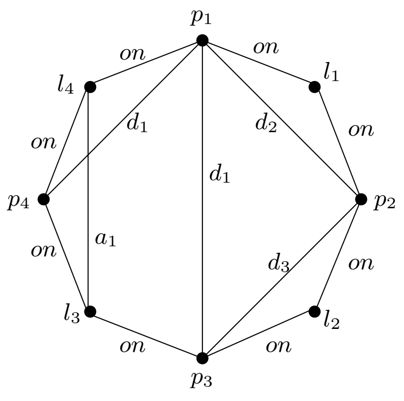


Figure 5.10: Geometric constraint graph built by the greedy algorithm.

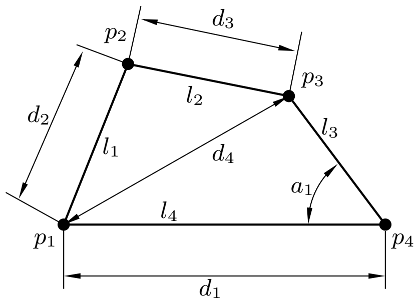


Figure 5.11: Solvable problem for the the graph in Figure 5.10.

must also fulfill an additional property: it must be solvable by a constructive geometric constraint solving technique.

We have reformulated the maximal conditional completion problem as a combinatorial optimization problem associated with a subset system. In this context, we have investigated the applicability of the greedy algorithm. Although we have shown that the greedy algorithm not always yields a solution to the maximal conditional completion problem, the experimental results reveal that the greedy algorithm can be successfully used to compute conditional completions.

We have also considered the maximal well constrained completion problem. Here, we only require the resulting graphs to be well constrained. Recalling standard results in matroid theory and combinatorial rigidity theory, we have shown that the greedy algorithm solves this problem.

Finally, we have presented three applications of the maximal conditional completion problem. First, we have shown how the maximal conditional completion problem can be used in the solution of the schema reconciliation problem in cooperative design. This problem originally inspired the study of the maximal conditional completion and it is studied in Chapter 6. Second, we have presented a generalization of the maximal conditional completion problem in which the set of additional constraints is structured in priority levels. And third, we have discussed the applicability of the maximal conditional completion to deal with over constrained geometric constraint problems.

6 Definition of a Multiple Views Model

A key component in computer aided design systems is the model that captures the shape design. The need for integration in concurrent cooperative design environments has lead to the integration of different product information domains resulting in what is known as the *product master model*, a single repository where all relevant product data resides. The application of the master model concept is growing rapidly, but there are still significant technical problems that need further study and research.

To support concurrent engineering, modifications required by an application should be introduced in the application's view where the need arises, and modifications in any view should be propagated automatically to all other views, see Bronsvoort *et al.* [5, 13, 14]. Therefore, maintaining consistent views is a central problem in research on product design and manufacture.

Current approaches handle the consistency problem by organizing the system as a one-way architecture. The object in an application view is derived from the object in a privileged view, usually the design view. The designer defines this view and conversion modules generate application-dependent views. If a modification is required, the privileged view must be edited and the updated views can be derived again, see Cunningham [11].

In this chapter, we develop a framework to support geometric constraint-based design systems with multiple views for concurrent engineering. The framework is based on a conceptual architecture with a master view and several client views with a two-way flow information between the master and client views. The framework addresses especially the problem of maintaining consistency between different views.

6.1 Views

We define a *view* as a geometric model in the sense of Definition 2.4. Although views are ordinary geometric models, it is usual to talk about views when we have a collection of models all of them referred to the same object. We say that all those models are different views of the same object.

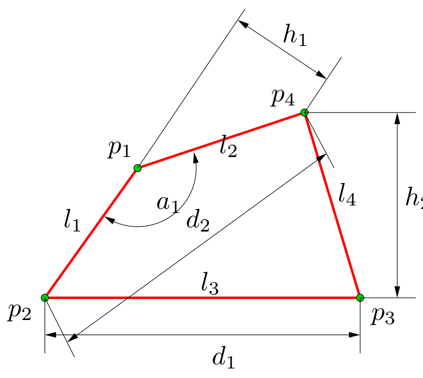
In this chapter we use the concepts of Chapter 2 when referring to views. We assume that we are using a particular ideal solver, according to Definition 2.3. We note a view as $W = \langle A, \alpha, \iota \rangle$ following the same notation used for geometric models. Here, A denotes an abstract problem, α a parameter assignment and ι an index assignment. When the solver is applied to A we obtain a constructive sequence S_A. Consequently, ι is an index of $V(\alpha.S_A)$. Because the solver is ideal, $V(\alpha.S_A) = V(\alpha.A)$ and thus realizations of $\alpha.A$ and indexed anchors of $V(\alpha.S)$ are the same elements. Therefore, we can say that W defines a realization or, equivalently, an indexed anchor. We note as r_W the realization (or the equivalent indexed anchor) defined by W.

Let $W = \langle A, \alpha, \iota \rangle$ a view. Then, it is useful to classify the constraints of the abstract problem A, in two subsets. The first defines the topology, and the second defines the set of geometric (or metric) constraints.

Definition 6.1 *We will say that two views of the same object are* compatible *if their corresponding abstract problems are built on the same set of geometric elements and have the same topology.*

Figures 6.5 and 6.6 show two compatible views. We are only interested in compatible views. Notice that two compatible views differ at most in the metric constraints, the index assignment and the parameter assignment.

In general we represent views in a textual way. For example, see Figure 6.1. Each view has an identifier plus six different sections. The first section lists the set of parameters, the second the set of geometric elements, the third defines the topology, and the fourth section defines the set of geometric constraints. The last two sections list de parameters and index assignment respectively.



```
view M
  param
    $d_1, d_2, a_1, h_1, h_2$ : real
  endparam
  index
    $s_1, s_2, s_3$ : sign
  endindex
  geom
    $p_1, p_2, p_3, p_4$ : point
    $l_1, l_2, l_3, l_4$ : line
  endgeom
  topology
    $onPL(p_2, l_1)$
    $onPL(p_4, l_2)$
    $onPL(p_2, l_3)$
    $onPL(p_4, l_4)$
    $onPL(p_1, l_1)$
    $onPL(p_1, l_2)$
    $onPL(p_3, l_3)$
    $onPL(p_3, l_4)$
  endtopology
  constraints
    $distPP(p_1, p_3, d_1)$
    $distPP(p_2, p_4, d_2)$
    $distPL(p_4, r_1, h_1)$
    $distPL(p_4, r_3, h_2)$
    $angleLL(r_1, r_2, a_1)$
  endconstraints
  param-assign
    $d_1 = 440.0$
    $d_2 = 440.0$
    $h_1 = 150.0$
    $h_2 = 250.0$
    $a_1 = 2.20$
  endparam-assign
  index-assign
    $s_1 = 1$
    $s_2 = 1$
    $s_3 = -1$
  endindex-assign
endview
```

Figure 6.1: A geometric constraint-based model.

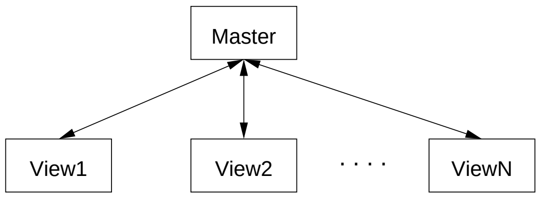


Figure 6.2: Master view architecture with client views.

6.2 The Multiple Views Model

We assume the master model scenario described by Hoffmann and Joan-Arinyo in [25, 26]. There is an object-oriented master view server that records all the information to be shared among the different views. The clients that connect the master view server are assumed to be autonomous but collaborating and they pledge to follow the protocols of the master view server.

The master view receives notifications from a client view that wishes to edit the object under design, according to its own application. The client then transmits the proposed changes to the master view server. The master view processes the changes and if it can successfully update, the master view commits to the change and the edit proposed by the client is successful.

Figure 6.2 illustrates the model of the overall architecture. This model allows synchronizing a client with the master view and vice versa. Synchronization between different client views must be performed through the master view. We assume that every item in the master view is placed by a client in charge of doing the primary editing of the view.

6.2.1 Protocols

Information that has been created and is owned by an specific client view can affect many other client views. This cross-dependency of information between views requires explicit protocols which ensure that information is added and updated maintaining consistency between views. We have designed a set of operations to meet the needs of the master view architecture with a two-way information flow. The operations are inspired on the

Berliner's CVS model [3]. Using these operations several protocols can be implemented. The set of operations is:

1. *OpenView*: This operation allows the user to open a new view. Let M be the master view. Then the operation defines a new view M' by replicating the geometry and the topology of M. Notice that the new view M' and the original one M are compatibles.

2. *Complete*: Let V be a completable view. To complete V means to transform it into a solvable view by adjoining extra constraints. We consider two different ways of completing a view: *conditional completion* and *free completion.* These operations have been introduced in Chapter 5. Note that in conditional completion we need an additional set of constraints over the same geometric elements. This means that to apply a conditional completion to V we need a compatible view V' that plays the role of additional set of constraints.

3. *ValuesTransfer*: Let V and V' be two compatible views of a given object. The *ValuesTransfer* operation assigns to the parameters of the constraints in a view V' values that are either explicitly represented in another view V or that can be computed from the actual geometry of V.

   When values are transferred from a client view V to the master model M, the operation is denoted as *Commit.* When values are transferred from M to the view V, the operation is denoted as *Update.*

Using this set of operations, a number of protocols can be developed. In what follows we illustrate one of them. See Figure 6.3. Assuming that a master model M is available, a new client view V is opened by issuing an *OpenView* operation. The new view V does not include metric constraints but has the same geometry and topology as M. Then, the user can add new constraints to V according to the specific application's view. Note that the resulting view V' does not need to be well constrained. Next, a *Complete* operation completes the view by adjoining constraints, in this example taken from the master view. The shape under design can be changed by changing the values assigned to the constraint parameters. When the editing is over, the master model M is updated by a *Commit* operation.

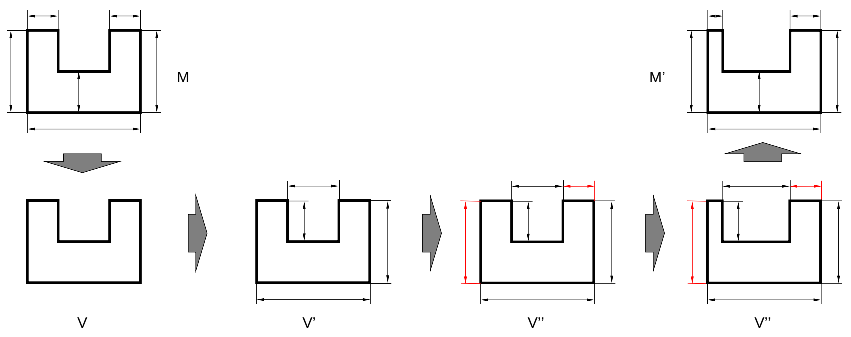


Figure 6.3: A work pattern using views.

Clearly, this is not the only possible pattern, but illustrates how the proposed operations can be used.

6.2.2 Master Invariant

To guarantee the master view coherence, we associate a first order predicate $\mathcal{P}$ to the master view M. The predicate should be defined on the geometric elements and parameters' values of M. Every time a *commit* operation is triggered to update the master view M, the predicate $\mathcal{P}$ is checked and the *commit* operation is actually carried out only if the master invariant holds for the updated master view. Similar mechanisms can be found in database technology, see Date's book, [12]. We call this mechanism the *Master invariant.*

For example, to guarantee that the slot in the object of Figure 6.4 is always part of the object's shape, we can define the master invariant labeled inv_1 as $\mathcal{P} \equiv dist(p_1, p_2) > 1$.

This mechanism can be easily extended to enforce the master invariant not only during commit time but also during the edition time of the view. This extension would be preferred when using long edition transactions.

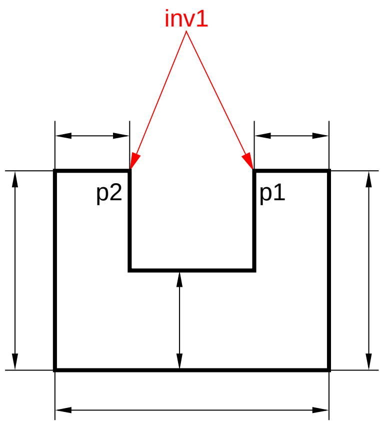


Figure 6.4: A master invariant example.

6.3 Implementation Issues

Implementing the *OpenView* operation is simple. All what is needed is to replicate the geometry and topology of some other already defined view, usually the master view.

Implementing *Completion* means to transform an under constrained view into a well constrained one by adjoining geometric constraints. Completion can be performed applying the techniques reported in Chapter 5.

Conceptually, *Commit* and *Update* are distict operations. However, both can be implemented with a *ValueTransfer* operation. The implementation of this operation is the objective of what follows.

6.3.1 ValuesTransfer Implementation

First we precisely define the *ValuesTransfer* operation:

Definition 6.2 *Let* $V = \langle A_V, \alpha_V, \iota_V \rangle$ *and* $W = \langle A_W, \alpha_W, \iota_W \rangle$ *be two different views. To* transfer the values *of* W *to* V *means to compute a new view* $V' = \langle A_V, \alpha_{V'}, \iota_{V'} \rangle$ *such that* $r_{V'} = r_W$*. This operation is written as* $Tr(W, V) = V'$*.*

Note that V and V' have the same abstract system and hence the same constraints and parameters.

The implementation of *ValueTransfer* is performed in two steps. If the views are $V = \langle A_V, \alpha_V, \iota_V \rangle$ and $W = \langle A_W, \alpha_W, \iota_W \rangle$, the valueTranfer $V' = Tr(W, V)$ can be performed as follows:

1. Using the solver compute the realization r_W from the view W. Since we assume a correct solver, $r_W \in V(\alpha_W.A_W)$. ι_W, the index assignment for W, indexes the set $V(\alpha_W.A_W)$.

   Now assign proper values to the parameters of V'. That is, compute a parameter assignment for V'. Notice that the parameters and constraints of V' are the same parameters of V.

   If a constraint in V is explicitly defined in W, the value of the parameter in W is assigned to the corresponding parameter in V'. If the constraint in V is not in W, the parameter value is *measured* in r_W and transferred to the parameter in V'.

   For example assume that in the view V there is a distance constraint like $dist(p_i, p_j) = d$ which is not in W. The value assigned to the parameter d is the value resulting from computing the distance from p_i to p_j in the realization r_W.

   The result of this computation is the parameter assignment of V', namely $\alpha_{V'}$.

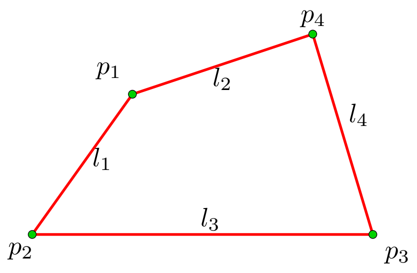
2. Now the index assignment of V', namely $\iota_{V'}$, must be computed.

   Notice that we know $\alpha_{V'}$ and thus $V(\alpha_{V'}.A_V)$ is also known. Since r_W is constructible and the solver is ideal $r_W \in V(\alpha_{V'}.A_V)$. The index assignment of this indexed anchor is computed by using an anchor based index selector, see Section 2.5.2. Luzón in [44] shows some techniques to efficiently implement index selectors.

This procedure yields a parameter assignment $\alpha_{V'}$ and an index assignment $\iota_{V'}$. Now, the new view resulting from transfering values is

$$V' = Tr(W, V) = \langle A_V, \alpha_{V'}, \iota_{V'} \rangle$$

.



```
view V
  geom
    $p_1, p_2, p_3, p_4$ : point
    $l_1, l_2, l_3, l_4$ : line
  endgeom
  topology
    $onPL(p_1, l_1)$
    $onPL(p_1, l_2)$
    $onPL(p_3, l_3)$
    $onPL(p_3, l_4)$
    $onPL(p_2, l_1)$
    $onPL(p_4, l_2)$
    $onPL(p_2, l_3)$
    $onPL(p_4, l_4)$
  endtopology
endview
```

Figure 6.5: The new client view opened from the master view.

6.4 Case Study

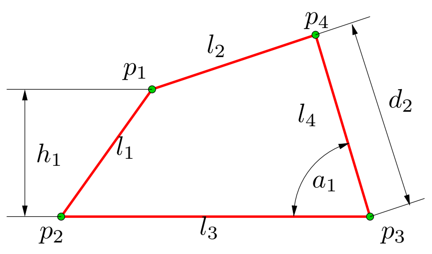
To illustrate how a multiple view model works we develop a simple case study with two different views: A master and a client view. Assume that the master view, M, is that depicted in Figure 6.1.

A new client view, compatible with M, can be opened with an *OpenView* operation. See Figure 6.5. Since the new client view is by definition under constrained, the geometric elements in the client view are placed using the coordinates of the master view.

Now the user adjoins to the client view the constraints of interest, for example, those depicted in Figure 6.6. In this example, the resulting client view, denoted V', still is under constrained.

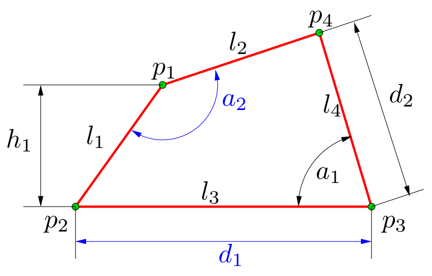
The next step is to transform the under constrained client view V' into a well constrained one by issuing a *complete* operation. Figure 6.7 shows the result of a conditional completion where the extra constraints are taken from the master view, M.

Now, the user changes the parameters value of view V''. If the set of values coherently define the object, the geometric constraint solver in the system



```
view V'
  param
    $d_2, a_1, h_1$ : real
  endparam
  geom
    $p_1, p_2, p_3, p_4$ : point
    $l_1, l_2, l_3, l_4$ : line
  endgeom
  topology
    $onPL(p_1, l_1)$
    $onPL(p_1, l_2)$
    $onPL(p_3, l_3)$
    $onPL(p_3, l_4)$
    $onPL(p_2, l_1)$
    $onPL(p_4, l_2)$
    $onPL(p_2, l_3)$
    $onPL(p_4, l_4)$
  endtopology
  constraints
    $distPL(p_1, l_3, h_1)$
    $distPP(p_3, p_4, d_2)$
    $angleLL(l_3, l_4, a_1)$
  endconstraints
endview
```

Figure 6.6: The new client view with some adjoined constraints.



view V"
  param
    d_2, a_1, h_1, a_2, d_1 : **real**
  endparam
  geom
    p_1, p_2, p_3, p_4 : **point**
    l_1, l_2, l_3, l_4 : **line**
  figeom
topology
    $onPL(p_1, l_1)$
    $onPL(p_1, l_2)$
    $onPL(p_3, l_3)$
    $onPL(p_3, l_4)$
    $onPL(p_2, l_1)$
    $onPL(p_4, l_2)$
    $onPL(p_2, l_3)$
    $onPL(p_4, l_4)$
endtopology
constraints
    $distPP(p_2, p_3, d_1)$
    $distPP(p_3, p_4, d_2)$
    $distPL(p_1, l_3, h_1)$
    $angleLL(l_1, l_2, a_2)$
    $angleLL(l_3, l_4, a_1)$
endconstraints
endview

Figure 6.7: The new well constrained client view.

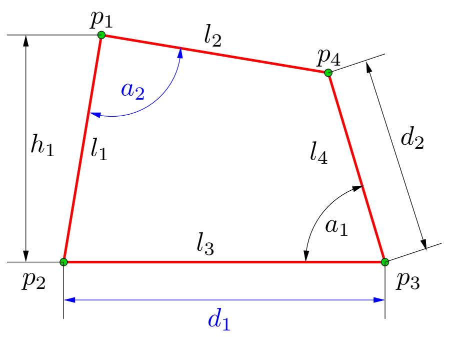


Figure 6.8: The new client view after changing the values of parameters h_1 and a_2.

generates the corresponding shape instance. Figure 6.8 shows the same object as Figure 6.7 after changing the value of parameters h_1 and a_2. If the set of parameters' values do not define a feasible object, the solver discards the construction and displays a warning.

Next, to update the master view, a *commit* operation is triggered. Since constraints with parameters a_1, h_1, d_1 and d_2 are common to the master view M and to the client view V, their values in M are directly updated. Constraint $dist(p_4, l_3) = h_2$ in M is not defined in V''. Therefore, the value is computed in V'' as the perpendicular distance from point p_4 to the straight line l_3. Changes introduced in the client view V'' have been propagated to the master model M, as a result they are synchronized. The resulting updated master view M is shown in Figure 6.9.

6.5 Summary

We have presented a set of tools and procedures that allow to implement multiple view geometric models in constraint-based geometric modeling systems. The tools are built on top of three basic operations *OpenView*, *Completion* and *ValuesTransfer* plus the *Master invariant*. These tools are of symbolic nature and yield robust mechanisms that allow to maintain consistency between different views of an object under design. In a case study we have illustrated how the tools we propose here solve the information flow needed to maintain consistency between views.

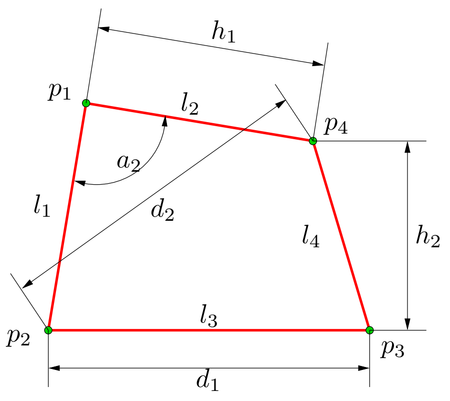


Figure 6.9: The master view after synchronization with the client view.

7 Conclusions and Future Work

In this chapter, we summarize the main achievements of this PhD thesis and point out directions for future work.

7.1 Conclusions

In this dissertation we have studied three main aspects concerning constructive geometric constraint solvers:

1. We have defined an architecture that is suitable to constructive geometric constraint solvers. This definition introduces three functional units which are defined in terms of their input and output. Every functional unit have a particular type of input and output data. All these data types have been identified and characterized. The proposed architecture has interest from a theoretical point of view. Moreover, the architecture can be translated to a software implementation and thus it have also an important practical interest.

2. We have studied the domain of the three constructive analyzers reported in [21, 50]. We have proved that the three analyzers have the same domain. This domain have been characterized by using the tree decomposition of graphs.

   The characterization of the domain of analysis algorithms has lead to the following additional results:

   - *Owen's* algorithm has been studied in depth and a new version of the algorithm described in [50] is given. This new version is easier to understand and clearly shows the divide-and-conquer structure of Owen's algorithm otherwise obscured.

- *Tree decompositions* are an interesting tool to understand analysys algorithms. The three analysis algorithms studied are in fact distinct algorithms to compute tree decompositions. Thus, tree decompositions are a tool useful to describe the domain of analysis methods and also an interesting tool to understand the methods themselves.

- The class of *solvable graphs* is defined as a recursive sequence similar to Henneberg sequences. Therefore, the domain of the studied analysis methods can be easily generated.

3. We have defined an operation to complete under constrained geometric constraint problems. We have studied which under constrained problems can be satisfactorily transformed into a solvable problem and then, the completion operation has been reformulated as a combinatorial optimization problem. We have studied the greedy algorithm on this problem. We have shown through extensive testing that, although it is suboptimal, it performs satisfactorily for real test cases.

   Based on this operation, along with the value transfer operation also defined in this work, we have stated a multiple views geometric constraint based model. This models allows to coherently maintain a set of views of the same object. The model defines a set of operations similar to those found on optimistic long transaction systems like CVS [3]. Using these operations a number of useful working protocols can be established. The resulting model is well suited to concurrent engineering environments.

7.2 Future Work

Several new problems have emerged during the development of this work. To our understanding, the most important are:

- Efficient versions of the three analysis methods studied here, [21, 50], are difficult to implement. Considering an analysis algorithm as a way to build a tree decomposition we think that simpler analysis algorithms can be devised while maintaining time efficiency. In particular, we think that the triconnectivity algorithm of Miller and Ramachandran [48] can be simplified to obtain a practical algorithm to compute the tree decomposition of a graph.

- A challenging problem is to devise an incremental method to compute a tree decomposition. That is, assume that we have a graph and its corresponding tree decomposition. When the graph is modified by edge addition or deletion, how can we incrementally compute the new tree decomposition (if it exists)?. Efficiently solving this problem will naturally lead to a new analysis algorithm. This algorithm would be very useful in real applications due to the interactive nature of geometric modelers.

- The greedy algorithm presented to solve the completion problem performs satisfactorily for real problems but is suboptimal. It would be desirable to prove or disprove whether the associated combinatorial optimization problem on the set of solvable geometric constraint problems is $\mathcal{NP}$-hard.

Bibliography

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Computer Science and Information Processing. Addison Wesley Publishing Company, Reading, MA, USA, 1974.

[2] Aldefeld, B. Geometric constraint solver. *Computer Aided Design 20*, 3 (Apr. 1988), 117–126.

[3] Berliner, B. CVS II: Parallelizing software development. `http://www.loria.fr/~molli/cvs-index.html`.

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

[5] Bronsvoort, W., and Jansen, F. Multi-view feature modelling for design and assembly. In *Advances in Feature Based Manufacturing*, J. Shah, M. Mäntylä, and D. Nau, Eds., Manufacturing Research and Technology, 20. Elsevier Science B.V., 1994, ch. 14, pp. 315–330.

[6] Brüderlin, B. Constructing three-dimensional geometric objects defined by constraints. In *Proceedings of the Workshop on Interactive 3D Graphics* (Oct. 1986), ACM, pp. 111–129.

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

[8] Brüderlin, B. Symbolic computer geometry for computer aided geometric design. In *Advances in Design and Manufacturing Systems, Proceedings of NSF Conference* (Tempe, AZ, Jan. 1990), NSF.

[9] Brüderlin, B. Using geometric rewrite rules for solving geometric problems symbolically. In *Theoretical Computer Science 116* (1993), Elsevier Science Publishers B.V., pp. 291–303.

[10] Chartrand, G., and Lesniak, L. *Graphs & Digraphs*, 3rd ed. Chapman & Hall, 1996.

[11] Cunningham, J., and Dixon, J. Designing with features. The origin of features. In *Computers in Engineering Conference and Exhibition* (San Francisco, 1988), V. Tipnis and E. Patton, Eds., vol. 1, ASME, pp. 237–243.

[12] Date, C. *An Introduction to Database Systems*, 7th ed. Addison-Wesley, 2000.

[13] de Kraker, K., Dohmen, M., and Bronsvoort, W. Multiple-way feature conversion to support concurrent engineering. In *Product Modeling for Computer Integrated Design and Manufacture* (London, UK, 1997), M. Pratt, R. Siriram, and M. Wozny, Eds., Chapman and Hall, pp. 203–212.

[14] Dohmen, M., de Kraker, K., and Bronsvoort, W. Feature validation in a multiple-view modeling system. In *16th ASME International Computers in Engineering Conference* (Irvin, NY, USA, Aug. 1996), ASME.

[15] Durand, C. *Symbolic and Numerical Techniques for Constraint Solving.* PhD thesis, Purdue University, Department of Computer Sciences, Dec. 1998.

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

[17] Even, S. *Graph Algorithms.* Computer Software Engineering Series. Computer Science Press Inc., Rockville, Maryland, USA, 1979.

[18] Fudos, I. Editable representations for 2D geometric design. Master's thesis, Department of Computer Sciences, Purdue University, 1993.

[19] Fudos, I. *Constraint Solving for Computer Aided Design.* PhD thesis, Purdue University, Department of Computer Sciences, 1995.

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

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

[22] Garling, D. *A Course in Galois Theory.* Cambridge University Press, 1986.

[23] Graver, J., Servatius, B., and Servatius, H. *Combinatorial Rigidity*, vol. 2 of *Graduate Studies in Mathematics.* American Mathematical Society, 1993.

[24] Gross, J., and Yellen, J. *Graph Theory and Its Applications.* Discrete Mathematics and its Applications. CRC Press, Boca Raton, FL, USA, 1999.

[25] Hoffmann, C., and Joan-Arinyo, R. CAD and the product master model. *Computer Aided Design 30*, 11 (1998), 905–918.

[26] Hoffmann, C., and Joan-Arinyo, R. Distributed maintenance of multiple product views. *Computer Aided Design 32*, 7 (June 2000), 421–431.

[27] Hoffmann, C., Lomonosov, A., and Sitharam, M. Decompostion Plans for Geometric Constraint Systems, Part I: Performance Measurements for CAD. *Journal of Symbolic Computation 31* (2001), 367–408.

[28] Hopcroft, J. E., and Tarjan, R. E. Dividing a graph into triconnected components. Tech. rep., Computer Science Department. Cornell University, Ithaca, NY. USA, Feb. 1974. New revision of TR 72-140.

[29] Jermann, C. *Résolution de Containtes Géométriques par Rigidification Récursive et Propagation d'Intervalles.* PhD thesis, Université de Nice - Sophia Antipolis, Dec. 2002. Written in French.

[30] Joan-Arinyo, R. Triangles, ruler and compass. Tech. Rep. LSI-95-6-R, Universitat Politècnica de Catalunya, Department LSI, 1995.

[31] Joan-Arinyo, R., Soto, A., Vila, S., and Vilaplana, J. A framework to support multiple views in geometric constrain-based models. In *Proceedings of the 8th. IEEE International Conference on Emerging Technologies and Factory Automation ETFA'2001* (Antibes-Juan les Pins, France, Oct. 2001), E. Dekneuvel, Ed., 8th. IEEE.

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

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

[34] Joan-Arinyo, R., Soto-Riera, A., and Vila-Marta, S. Tools to deal with under-constrained geometric constraint graphs. In *6th. Asian Symposium on Computer Mathematics. Session on Geometric Constraint Solving* (Beijing (China), Apr. 2003).

[35] Joan-Arinyo, R., Soto-Riera, A., Vila-Marta, S., and Vilaplana, J. On the domain of constructive geometric constraint solving techniques. In *Proc. of the Spring Conference on Computer Graphics* (2001), R. Ďurikovič and S. Czanner, Eds., IEEE Computer Society, pp. 49–54.

[36] Joan-Arinyo, R., Soto-Riera, A., Vila-Marta, S., and Vilaplana, J. On the completion of underconstrained geometric constraint problems. In *Proceedings of Third International NAISO Symposium on Engineering of Intelligent Systems* (Málaga (Spain), Sept. 2002), ICSC-NAISO, ICSC-NAISO Academic Press. ISBN: 3-906454-32-0.

[37] Joan-Arinyo, R., Soto-Riera, A., Vila-Marta, S., and Vilaplana, J. Revisiting decomposition analysis of geometric constraint graphs. *Computer Aided Design* (2003). In press.

[38] Joan-Arinyo, R., Soto-Riera, A., Vila-Marta, S., and Vilaplana-Pastó, J. Declarative characterization of a general architecture for constructive geometric constraint solvers. In *Proc. of the 3IA International Conference* (Limoges, France, May 2002), D. Plemenos, Ed., MSI, University of Limoges, pp. 55–69.

[39] Joan-Arinyo, R., Soto-Riera, A., Vila-Marta, S., and Vilaplana-Pastó, J. Revisiting decomposition analysis of geometric constraint graphs. In *Proc. of the Solid Modeling and Applications SM'02* (Saarbrüken, Germany, June 2002), K. Lee and N. Patrikalakis, Eds., ACM Press, pp. 105–115.

[40] Joan-Arinyo, R., Soto-Riera, A., Vila-Marta, S., and Vilaplana-Pastó, J. Transforming an under-constrained geometric constraint problem into a well-constrained one. Research LSI-02-69-R, Dept. Llenguatges i Sistemes Informàtics, UPC, Barcelona, Nov. 2002.

[41] Joan-Arinyo, R., Soto-Riera, A., Vila-Marta, S., and Vilaplana-Pastó, J. Transforming an under-constrained geometric constraint problem into a well-constrained one. In *Proc. of the Solid Modeling and Applications SM'03* (Seattle, WA (USA), June 2003).

[42] Klop, J. Term rewriting systems. In *Background: Computational Structures*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds., vol. 2 of *Handbook of Logic in Computer Science.* Clarendon Press, 1992, pp. 1–117.

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

[44] Luzón, M. *The Root Identification Problem in Constructive Geometric Constraint Solving.* PhD thesis, Universidade da Vigo, 2001. Writen in spanish.

[45] Manna, Z., and Waldinger, R. *The Deductive Foundations of Computer Programming.* Addison-Wesley Pu. Co., Reading, MA, 1993.

[46] Mäntylä, M. *Introduction to Solid Modeling.* Computer Science Press, Rockville, MD, 1988.

[47] Mata, N. *Constructible Geometric Problems with Interval Parameters.* PhD thesis, Universitat Politècnica de Catalunya, LSI Department, Barcelona, Spain, 2000.

[48] Miller, G. L., and Ramachandran, V. A new graph triconnectivity algorithm and its parallelization. *Combinatorica 12* (1992), 53–76.

[49] Mortenson, M. *Geometric Modeling.* Wiley, New York, 1985.

[50] Owen, J. Algebraic solution for geometry from dimensional constraints. In *Proc. of ACM Symposium on Foundations of Solid Modeling* (Austin TX, USA, June 1991), R. Rossignac and J. Turner, Eds., ACM, ACM Press, pp. 397–407.

[51] Owen, J. Constraints on simple geometry in two and three dimensions. *International Journal of Computational Geometry & Applications 6*, 4 (1996), 421–434.

[52] Owen, J., and Whiteley, W. Constraining plane geometric configurations in cad: Directions and distances. Manuscript, July 1996.

[53] Oxley, J. *Matroid Theory.* Oxford University Press, Oxford, UK, 1992.

[54] Papadimitriou, C. H., and Steiglitz, K. *Combinatorial Optimization: Algorithms And Complexity.* Dover Publications, May 1998.

[55] Requicha, A. Representations for rigid solids: Theory, methods, and systems. *ACM Computing Surveys 12*, 4 (Dec. 1980), 437–464.

[56] Servatius, B., and Whiteley, W. Constraining plane geometric configurations in cad: Combinatorics of directions and lengths. Manuscript, 1994.

[57] Solano, L. *Constructiva Solid Modeling Based on Constraints.* PhD thesis, Dept. Llenguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya, Barcelona, Spain, May 1999.

[58] Soto, A. *Satisfacció de Restriccions Geomètriques en 2D.* PhD thesis, Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona, Spain, 1998. Written in Catalan.

[59] Todd, P. A k-tree generalization that characterizes consistency of dimensioned engineering drawings. *SIAM J. Discrete Mathematics 2*, 2 (1989), 255–261.

[60] van Leeuwen, J. Graph algorithms. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier Science Publishers B.V., 1990, ch. 10, pp. 527–631.

[61] Verroust, A. *Etude de Problèmes Liés à la Dèfinition, la Visualisation et l'Animation d'Objects Complexes en Informatique Graphique.* PhD thesis, Universite de Paris-Sud, Centre d'Orsay, 1990.

[62] Verroust, A., Schonek, F., and Roller, D. Rule-oriented method for parameterized computer-aided design. *Computer Aided Design 24*, 10 (Oct. 1992), 531–540.

[63] Whiteley, W. Matroids and rigid structures. In *Matroid Applications*, N. White, Ed., vol. 40 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, Great Britain, 1992, ch. 1, pp. 1–53.

[64] Whiteley, W. Rigidity and scene analisys. In *Handbook of Discrete and Computational Geometry*. CRC Press, 1997, pp. 893–916.