Department of Computer Science

PhD Program in Computing

Doctoral Dissertation

Direct Tree Decomposition of Geometric Constraint Graphs

Marta I. Tarrés-Puertas

Barcelona, 2014

Advisors:
Robert Joan-Arinyo, Sebastià Vila-Marta

Abstract

The evolution of constraint based geometric models is tightly tied to parametric and feature-based Computer-Aided Design (CAD) systems. Since the introduction of parametric design by Pro/Engineer in the 1980's, most major CAD systems adopted constraint based geometric models as a core technology. Constraint based geometric models allowed CAD systems to provide a more powerful data model while offering an intuitive user interface. Later on, the same models also found application to fields like linkage design, chemical modeling, computer vision and dynamic geometry.

Constraint based geometric models are unevaluated models. A key problem related to constraint based geometric models is the geometric constraint based solving problem which, roughly speaking, can be stated as the problem of evaluating a constraint based model. Among the different approaches to geometric constraint solving, we are interested in graph-based Decomposition-Recombination solvers. In the graph-based constructive approach, the geometric problem is first translated into a graph whose vertices represent the set of geometric elements and whose edges are the constraints. Then the constraint problem is solved by decomposing the graph into a set of sub-problems, each sub-problem is recursively divided until reaching basic problems which are solved by a dedicated equational solver. The solution to the initial problem is computed by merging the solutions to the sub-problems.

The approach used by DR-solvers has been particularly successful when the decomposition into subproblems and subsequent recombination of solutions to these subproblems can be described by a plan generated a priori, that is, a plan generated as a preprocessing step without actually solving the subsystems. The plan output by the DR-planner remains unchanged as numerical values of parameters change. Such a plan is known as a DR-plan and the unit in the solver that generates it is the DR-planner. In this setting, the DR-plan is then used to drive the actual solving process, that is, computing specific coordinates that properly place geometric objects with respect to each other.

In this thesis we develop a new DR-planner algorithm for graph-constructive two dimensional DR-solvers. This DR-planner is based on the tree-decomposition of a graph. The triangle- or *tree-decomposition* of a graph decomposes a graph into three subgraphs such that subgraphs pairwise share one vertex. Shared vertices are called *hinges*. The tree-decomposition of a geometric constraint graph is in some sense the construction plan that solves the corresponding problem. The DR-planner algorithm first transforms the input graph into a simpler, planar graph. After that, an specific planar embedding is computed for the transformed graph where hinges, if any, can be straightly found. In the work we proof the soundness of the new algorithm. We also show that the worst case time performance of the resulting algorithm is quadratic on

the number of vertices of the input graph. The resulting algorithm is easy to implement and is as efficient as other known solving algorithms.

Acknowledgements

I have no special talents. I am only passionately curious.

(Albert Einstein)

First and foremost, I would like to express my sincere gratitude to Dr. Robert Joan-Arinyo and Dr. Sebastià Vila-Marta for the continuous support of my Ph.D study and research, for his patience, motivation and enthusiasm. Every meeting with them was a highly fulfilling experience. The ideas that are presented in this work were born during those discussions.

Professor Robert Joan-Arinyo met all of our brainstorming with patient explanations and unmatched expertise. His sharp questions helped me to better understand the limitations of my research and motivated me to find better ways to explain my results and illustrate them. Thanks for your deeply rewarding cross-disciplinary collaboration.

The genesis of this thesis can be traced back when Professor Sebastià Vila brought me the opportunity to join the GIE group in 2007. During these years he was an invaluable source of feedback and support. I thank him for the wonderful collaboration, for all his patient advice, help and support on matters technical and otherwise, and for all the things I learned from him. He challenged me when I needed to be challenged and boosted my confidence when it needed to be boosted. He has been an incredible teacher, mentor guide and role model.

I thank the anonymous referees of Computer-Aided Design [71] for their helpful comments and suggestions that helped to improve the explanations, and thanks to the referees in [69] and [70] for their reviewed comments.

I warmly thank my colleagues of the Computer Science in Engineering (GIE) research group and my colleagues of the Department of Electronic Systems Design and Programming (DIPSE) for their companionship and encouragement.

On a more personal note, special thanks to my parents, who instilled in me the value of hard work, for their unwavering support throughout this and all my adventures. To my daughter and my son, for being my endless inspiration. My daughter helped to put my work in perspective and to improve it. My son was born part way through my thesis research, and he gave me added incentive to finish the manuscript and to continue to research. And last, to Daniel, for his inexhaustible sense of humor.

Contents

Abstract i

Acknowledgements iii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Goals . . . 3
- 1.3 Methodology . . . 3
- 1.4 Main Contributions . . . 4
  - 1.4.1 Scientific Publications . . . 4
- 1.5 Thesis Organization . . . 4

2 Graph-theoretic Background 7
- 2.1 Basic Definitions . . . 7
- 2.2 Operations on Graphs . . . 8
- 2.3 Connectivity of a Graph . . . 9
- 2.4 Trees and Fundamental Circuits . . . 10
- 2.5 Depth-First Search Algorithm . . . 13
- 2.6 Planar Graphs and Embeddings . . . 13
- 2.7 Components and Bridges . . . 14
- 2.8 Planar Graphs and Bridges . . . 16
- 2.9 Chapter Summary . . . 18

3 Geometric Constraint Solving Background 19
- 3.1 Geometric Constraint Problem . . . 19
  - 3.1.1 The General Geometric Constraint Problem . . . 19
  - 3.1.2 The Basic Geometric Constraint Problem . . . 21
  - 3.1.3 Rigidity Characterization . . . 21
- 3.2 Representation of Geometric Constraint Problems . . . 24
  - 3.2.1 Equational Representation . . . 24
  - 3.2.2 Logic-Based Representation . . . 25
  - 3.2.3 Graph-Based Representation . . . 26
- 3.3 Resolution Methods in GCS: An overview . . . 31
  - 3.3.1 Equational Methods . . . 32

3.3.2 Logic-Based Methods . . . 33
3.3.3 Graph-Based Approach . . . 34
3.4 Graph-Constructive Solvers: Overview and Background . . . 36
3.4.1 Decomposition-Recombination Solvers . . . 39
3.5 Tree Decomposition of a Geometric Constraint Graph . . . 40
3.5.1 Tree Decomposable Graphs . . . 45
3.6 Construction Plan . . . 45
3.7 Chapter Summary . . . 50

4 A new DR-planner 51
4.1 Overview . . . 51
4.2 Decomposing 0-connected Graphs . . . 51
4.3 Decomposing 1-connected Graphs . . . 54
4.4 Decomposing Biconnected Graphs . . . 54
4.4.1 Decomposing Graphs with Degree 2 Vertices . . . 54
4.4.2 Decomposing Graphs with Higher Degree Vertices . . . 55
4.4.3 The Algorithm . . . 57
4.4.4 Computing the Set of Fundamental Circuits . . . 58
4.4.5 Computing the Set of Bridges . . . 59
4.4.6 Computing the Collapsed Graph . . . 59
4.4.7 Computing the Merged Graph . . . 62
4.4.8 Computing the Planar Embedding of the Merged Graph . . . 64
4.4.9 Computing the Hinges . . . 65
4.4.10 Tree Decomposition . . . 66
4.5 Case study . . . 67
4.5.1 Computing the Set of Fundamental Circuits . . . 68
4.5.2 Computing the Set of Bridges . . . 71
4.5.3 Computing the Collapsed graph . . . 71
4.5.4 Computing the Merged Graph . . . 74
4.5.5 Computing the Planar Embedding of the Merged Graph . . . 74
4.5.6 Computing the Hinges . . . 74
4.5.7 Recursive Tree Decomposition Algorithm . . . 74
4.6 Experimental Runtime Behavior . . . 76
4.7 Chapter Summary . . . 77

5 Correctness Proof 79
5.1 Hinges and Fundamental Circuits . . . 79
5.2 The Algorithm Preserves Hinges . . . 81
5.3 The Merged Graph is Planar . . . 83
5.4 Decomposability . . . 84
5.5 Chapter Summary . . . 86

6 Complexity 87
6.1 Graph-connectivity . . . 87
6.2 Complexity of Decomposition of 0-connected Graphs . . . 88

6.3 Complexity of Decomposition of 1-connected Graphs . . . 89
6.4 Complexity of Decomposition by Deleting degree 2 vertices . . . 89
6.5 Complexity of Decomposition of Biconnected Graphs . . . 89
6.5.1 The Set of Fundamental Circuits . . . 90
6.5.2 The Set of Bridges Implementation . . . 90
6.5.3 The Collapsed Graph . . . 90
6.5.4 The Merged Graph . . . 91
6.5.5 The Merged Graph Embedding Implementation . . . 95
6.5.6 Computing the Hinges . . . 96
6.5.7 The Basic Tree-decomposition Algorithm . . . 97
6.6 The Tree-decomposition DR-Planner Algorithm Running Time . . . 97
6.7 Chapter Summary . . . 98

7 Conclusions and Future Research Directions 105

Bibliography 109

Case Study Decomposition Examples 121
1 Case Study Decomposition: A 2D cross section. . . . 121
1.1 Computing the Set of Fundamental Circuits . . . 121
1.2 Computing the Set of Bridges . . . 124
1.3 Computing the Collapsed graph . . . 126
1.4 Computing the Merged Graph . . . 126
1.5 Computing the Planar Embedding of the Merged Graph . . . 128
1.6 Computing the Hinges . . . 128
2 Case Study Decomposition: A theoretical example . . . 130
2.1 Computing the Set of Fundamental Circuits . . . 130
2.2 Computing the Set of Bridges . . . 131
2.3 Computing the Collapsed Graph . . . 132
2.4 Computing the Merged Graph . . . 133
2.5 Computing the Planar Embedding of the Merged Graph . . . 133
2.6 Computing the Hinges . . . 135
2.7 Recursive Tree Decomposition Algorithm . . . 135
3 Case Study Decomposition: A mechanism . . . 135

List of Figures

2.1 Graph example. 8
2.2 A disconnected graph. 9
2.3 A 0-connected graph. 10
2.4 A 1-connected graph. 10
2.5 (a) A graph G, (b) A tree T of G and (c) A co-spanning tree T^* of T. 11
2.6 Fundamental circuits of a graph G. 12
2.7 (a) G, (b) one of its DFS trees and (c) the co-spanning tree. 13
2.8 Planar embedding of a graph. 14
2.9 G, components induced by K_1 and bridges 15
2.10 (a) Graph. (b) Bridges. 16
2.11 (a) Bridges interlacement case 1. (b) Bridges case 2. 17
2.12 Interlacing bridges B_1 and B_2. 18

3.1 A general geometric constraint solving problem in 2D. 21
3.2 An structurally well-constrained problem. 22
3.3 An structurally under-constrained problem. 23
3.4 An structurally over-constrained problem. 23
3.5 A geometric constraint problem. 24
3.6 Equations for the problem in Figure 3.5. 25
3.7 Logic-based representation for the problem in Figure 3.5. 25
3.8 Graph representation of the problem in Figure 3.5. 26
3.9 A geometric constraint graph example. 27
3.10 A frontal view of an elevator door actuator mechanism. 28
3.11 A geometric constraint graph of the problem in Figure 3.10. 29
3.12 Geometric constraints for the problem in Figure 3.10. 30
3.13 A geometric constraint graph representation for the mechanism in Figure 3.11. 30
3.14 Crank and connecting rod problem and mechanism. 31
3.15 A geometric constraint graph for the problem of Figure 3.14. 32
3.16 Architecture data flow diagram. 38
3.17 (a) Set S with three or more members. (b) Set decomposition of S. 41
3.18 (a) G. (b) A non-decomposition set of G (broken edge). 42
3.19 A Geometric constraint problem. 42
3.20 Graph representation and graph decomposition of the problem in Figure 3.19. . 42
3.21 Tree decomposition of graph in Figure 3.20a. 44

3.22 Recursive decomposition of graph G in Figure 3.20a. 44
3.23 Classification of geometric constraint graphs, [67]. 45
3.24 Construction of a part of the plan of the tree decomposition in Figure 3.21. . . 46
3.25 Peaucellier's linkage. 47
3.26 Geometric constraints for Peaucellier's linkage in Figure 3.25. 48
3.27 Geometric Constraint graph for Peaucellier's linkage in Figure 3.25. 48
3.28 Construction plan for Peaucellier's linkage in Figure 3.25. 49

4.1 Decomposition of 0-connected graphs. 52
4.2 An example of 0-connected graph decomposition 53
4.3 An example of 0-connected graph decomposition 53
4.4 Decomposition of 1-connected graphs. 54
4.5 A 1-connected graph decomposition 55
4.6 Decomposition based in a 2-degree vertex. 55
4.7 Decomposition by deleting degree 2 vertices 56
4.8 Case study graph. 59
4.9 (a) Spanning tree of the case study graph and (b) Associated circuits. 60
4.10 (a) Circuit C_1 of the case study graph and (b) Induced bridges B_1, B_2, B_3. . . . 60
4.11 (a) Bridges. (b) Star Bridges. 61
4.12 (a) Graph (b) Collapsed graph. 61
4.13 (a) Bridges (b) Collapsed Graph. 62
4.14 Bridges example sharing 3 attachments 63
4.15 Collapsed graph example with stars S_1 and S_2 that interlace. 63
4.16 (a) Graph and (b) Collapsed graph with several interlacing stars. 64
4.17 Collapsed graph example with stars S_1 and S_2 that do not interlace. 64
4.18 Collapsed graph example with stars S_1 and S_2 that do not interlace. 65
4.19 (a) Collapsed graph. (b) Merged graph. 66
4.20 A circuit embedding 66
4.21 Labeling of circuits and bridges. 67
4.22 Planar embedding for the merged graph given in Figure 4.19b). 67
4.23 Graph and clusters G_1, G_2 and G_3. 68
4.24 Collection of set decompositions for the case study graph in Figure 4.8. 69
4.25 A bridge coplanar compound truss example. 70
4.26 Bridge coplanar compound truss. 70
4.27 (a) Truss constraint graph (b) Spannig tree. 70
4.28 Fundamental Circuits Computation 71
4.29 Classes induced by the circuit C_5, (thick line) in the truss graph. 72
4.30 Components defined by the classes in the truss graph for the circuit C_5. 72
4.31 Bridges (thin line) induced in the truss graph by circuit C_5. 73
4.32 Truss graph example. (a) Collapsed Graph (b) Merged Graph. 73
4.33 Planar embedding for the running merged truss graph. 74
4.34 Decomposition of the truss graph 75
4.35 Recursive treedecomposition of the truss graph G. 75
4.36 Behavior of the algorithms A_1, A_2, A_3, and A_4 on the dataset D_1. 76
4.37 Behavior of the algorithms A_1, A_2, A_3, and A_4 on the dataset D_2. 77

5.1 Circuits in a set decomposition of a biconnected graph. 80
5.2 Circuit segments induced by hinges 82
5.3 Planar embedding of a fundamental circuit, hinges and clusters 84
5.4 Clusters embedding 85
5.5 Planar embedding where faces f_i and f_j share vertices $\{a, b, c\}$. 85

6.1 Examples of bridges that do not interlace. 94
6.2 Merging bridges example with one bridge subsumed. 94
6.3 Labeling of circuit and bridges attachments. 95

1 $2D$ cross section parametrically defined by geometric constraints. 122
2 Constraint Graph for the $2D$ cross section in Figure 1. 123
3 Geometric constraint graph after deleting degree 2 vertices 123
4 Spanning Tree T of G. 124
5 Fundamental Circuits Computation 125
6 Classes induced by the circuit C_6, (thick line) in the graph. 126
7 Components defined by the classes in the graph for the circuit C_6. 127
8 Bridges (thin line) induced in the constraint graph by circuit C_6. 127
9 Collapsed Graph. 128
10 Merged Graph. 128
11 Embedding of the merged graph 129
12 Graph G and clusters G_1, G_2 and G_3. 129
13 Graph G. 130
14 Calculation of DFS tree of G. 131
15 G and fundamental circuit C_{11}. 132
16 (a) The collapsed graph G' (b) The merged graph G''. 133
17 Planar graph G'''. 134
18 Graph G and clusters G_1, G_2 and G_3. 134
19 Recursive tree-decomposition of G_1. 135
20 Watt's linkage in a car suspension (Source: Wikipedia). 135
21 Watt's linkage diagram (Source: Wikipedia). 136
22 Geometric Constraint graph associated to the Watt's mechanism. 137
23 One decomposition step exemplification 137

List of Algorithms

1 Exhaustive searh of hinges in a graph. 43

2 The general new constructive tree-decomposition based *DR-planner*. 52
3 The basic tree-decomposition algorithm for the DR-planner. 58

4 Decomposition of 0-connected graphs. 88
5 Decomposition of 1-connected graphs. 89
6 Computing the set of bridges. 91
7 Algorithm to compute the collapsed graph. 92
8 The merged graph implementation 93
9 Dispaching internal edge procedure of the merging algorithm 99
10 Dispaching first edge procedure of the merging algorithm 100
11 Sorting the set of edges of the merged graph implementation. 101
12 The embedding of the merged graph implementation 102
13 Algorithm to find the hinges. 103

CHAPTER 1

Introduction

> Whatever you can do, or dream you can, begin it. Boldness has genius, power, and magic in it.
>
> *(Johann Wolfgang von Goethe)*

Geometric models are data structures designed with the aim to represent the physical properties of objects and are intended to represent a unique solid. The main properties encoded include geometric and topological characteristics of physical objects.

Computer Aided Design (CAD) systems are software applications built to help industrial designers in 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. Geometric modelling began in the mid 1970's with two competing approaches, Constructive Solid Geometry (CSG) and Boundary Representation (Brep). Over time Brep became the dominant representation for CAD models. Later on constraint-based geometric models were introduced. The most salient characteristic of a constraint-based geometric model is the ability to describe a family of physical objects. Because of this characteristic, given a constraint-based model we can instantiate it by giving values to a well defined set of parameters. The result is an actual geometric model, usually a Brep, that belongs to the described family.

Parametric CAD/CAM systems capture the design intent by defining functional relationships between dimensional variables and geometric elements, that is, by applying constraint-based geometric design. Proengineer, [22], is an example of a commercial CAD system that uses parametric design. Constraint-based geometric models are at the core of such systems.

A paramount issue found in constraint-based geometric design is the constraint solving problem which can be roughly summarized as follows:

> Given a set of geometric elements and a set of constraints between them, place each geometric element in such a way that the constraints are fulfilled.

We consider geometric constraint problems in the Euclidean plane consisting of a finite set of geometric objects and a finite set of constraints defined between them. The geometric objects are drawn from a fixed set of elements such as points, straight lines, circles and arcs

of circle. The constraints include topological constraints such as tangency, incidence and perpendicularity, and metric constraints such as point-point distance, perpendicular point-straight line distance and line-line angle.

Algorithms that solve geometric constraint problems are named *solvers*. Among the existing solving methods we focus on *constructive techniques* and more specifically on *Decomposition-Recombination* solvers. In these techniques the input is a geometric constraint problem represented as a geometric constraint graph. Then the graph is decomposed yielding as output a constructive plan, that is, a sequence of basic steps that describe how to build a solution to the constraint-based geometric problem. Basic steps correspond to elemental operations which are solved with dedicated algorithms.

1.1 Motivation

Following Hoffmann *el al.*, [45], the desirable properties of solvers include efficiency, simplicity, domain richness and soundness. The design of solvers that fulfill these properties exhibits some challenging issues. The research of efficient solvers is paramount since the early nineties. Many attempts to provide general, powerful and efficient constructive graph-based techniques have been reported in the literature. In Chapter 3 we overview the main approaches published so far.

Decomposition-recombination solvers (DR-solvers) apply a divide-and-conquer recursive pattern: first the problem is decomposed into simpler subproblems, the simpler subproblems are solved and then their solutions are recombined.

Our interest focuses on constructive DR-solvers where the geometric constraint problem is abstracted as a graph. The constructive DR-solvers output a constructive plan, that is, a sequence of basic steps that describe how to position geometric objects such that constraints are fulfilled. These steps are basic geometric operations, for example, ruler-and-compass operations.

The works of Joan-Arinyo *et al.*, [66–68, 128] defined the *tree decomposition* of a constraint graph. We will refer to decomposition trees in Chapter 3. Decomposition trees, despite have their origins in theoretical work to characterize the domain of certain solvers, see [65,68], allow to represent construction plans. In some sense if we can obtain a tree decomposition for a geometric constraint graph we have solved the corresponding geometric constraint problem.

The computation of a tree decomposition of a constraint graph can indirectly be done through known methods like that of Owen, [99]. However, it is highly desirable to have an algorithm that allows the direct calculation of the tree decomposition of a constraint graph. The main objective of this work is to develop a new general, correct and efficient DR-solver based on computing a tree decomposition of the input graph.

The work presented in this thesis has been developed in the framework of the research on geometric constraint solving undertaken by Professor Robert Joan-Arinyo and Professor Sebastià Vila-Marta and the Engineering Computing Group (GIE). The GIE conducts research in computer graphics, solid modeling, and applications build on top of these technologies. This research has been partially supported by the Spanish Ministry of Science and FEDER grants TIN2004-06326-C03-01 and TIN2007-67474-C03-01, and by the Spanish Ministerio de Economia y Competitividad and FEDER grant TEC2012-35571.

1.2 Goals

The main goal of this work is to develop a new correct and efficient DR-solver based on computing a tree decomposition of the input graph. This involves:

- The design of an algorithm to compute a tree decomposition of a 2D geometric constraint graph. The algorithm should efficiently deal with well-constrained and under constrained problems.
- To prove the algorithm soundness.
- To analyze the worst case time complexity of the resulting algorithm.
- To gain better insight into the properties of tree decomposable graphs.

1.3 Methodology

Our research follows a method including both analytic processes and empirical experimentation with the goal of developing a new DR-planner algorithm according to the following steps:

1. Theoretical study of the existing methods in the geometric constraint solving research area.
2. Preliminary design of a new algorithm as a follow up of the works already developed in our research group, Joan-Arinyo *et al.*, [68, 114].
3. Design and verification of the algorithm as well as simulation of the algorithm runtime behavior. The new DR-planner algorithm was tested and the empirical behavior was documented in [70].
4. Development of a complete proof of correctness for the new algorithm in the sense that given a tree decomposable non over-constrained graph the new algorithm successfully decomposes it.
5. An efficient algorithm to identify interlacement of graph bridges was devised and tested.
6. An specific algorithm to figure out a planar embedding of a transformed constraint graph was developed.
7. An algorithm to compute a decomposition step was devised and its running time was analyzed.
8. Finally, a full recursive tree-decomposition based DR-planner was developed and its running time for the worst case was figured out.

1.4 Main Contributions

The main contributions of this work are the following:

- A new DR-planner based on tree decomposition has been developed. This algorithm decomposes graphs representing either under-constrained or well-constrained $2D$ geometric problems. This DR-planner belongs to the constraint shape recognition class in the sense of [45].

  The algorithm is inspired in the works of Miller and Ramachandran, [93], to compute triconnected components of a graph. The algorithm proceeds by exploring the fundamental circuits of the graph looking for a specific triple of vertices that allows to decompose the graph. These triples are named *hinges*. In order to find hinges the graph is transformed into an equivalent planar graph. The embedding of this planar graph allows to efficiently obtain the hinges if any.

- We prove that the algorithm is correct in the following sense: if the input graph is a tree-decomposable graph it is always decomposed, conversely, if the input graph is a non tree-decomposable graph the algorithm fails.

- We prove that the algorithm worst case time complexity is $O(n^2)$ where n is the number of geometric elements in the problem.

1.4.1 Scientific Publications

The scientific contributions of this thesis are the following,

1. JOAN-ARINYO, R., TARRÉS-PUERTAS, M., AND VILA-MARTA, S. Geometric constraint graphs decomposition based on computing graph circuits. In *Automated Deduction in Geometry - 7th International Workshop, ADG 2008, Shanghai, China* (2008), pp. 96–101, [69].

2. JOAN-ARINYO, R., TARRÉS-PUERTAS, M., AND VILA-MARTA, S. Treedecomposition of geometric constraint graphs based on computing graph circuits. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling* (New York, NY, USA, 2009), SPM'09, ACM, pp. 113-122, [70].

3. JOAN-ARINYO, R., TARRÉS-PUERTAS, M., AND VILA-MARTA, S. Decomposition of geometric constraint graphs based on computing fundamental circuits. Correctness and complexity. *Computer-Aided Design 52* (2014), 1-16, [71].

1.5 Thesis Organization

This thesis is structured in seven chapters. Chapter 1 presents the motivation and objectives of the undertaken research and briefly introduces the new DR-planner.

Chapter 2 provides the basic background concerning graphs theory needed to follow the rest of this work.

Chapter 3 introduces the background to geometric constraint problems and geometric constraint solving field, takes a closer look at the geometric constraint solving algorithms and their analysis, and reviews the state of the art in geometric constraint solving over the last decades. A comprehensive review of the published literature concerning geometric constraint solving is included focusing on constructive, graph-based decomposition-recombination solvers.

Chapter 4 is the core of this work and describes the new constructive Decomposition-Recombination planner developed.

In Chapter 5 we develop the correctness proof of our DR-planner algorithm.

Chapter 6 deals with the implementation details and the evaluation of the worst case efficiency of the new algorithm.

Chapter 7 sums up the contributions in this thesis and highlights open problems for future research.

Finally we include an Appendix with additional examples that illustrate how the new DR-planner works

CHAPTER 2

Graph-theoretic Background

> If I were again beginning my studies, I would follow the advice of Plato and start with mathematics.
>
> *(Galileo Galilei)*

In this Chapter we recapitulate some basic graph theory concepts that will be used in this manuscript. Most of the material in this chapter is standard. However, we present the basic concepts and their notation here. For more information on general graphs theory we refer to the books of Even, [28], Thulasiraman and Swamy, [123] and Gross *et al.*, [39].

We start with some basic definitions. The analysis of the rest of concepts are grouped into operations on graphs, connectivity of a graph, trees and fundamental circuits, Depth-First search algorithm, planar graphs and embeddings, components and bridges, and finally, planar graphs and bridges.

2.1 Basic Definitions

In this work, a *graph* $G = (V, E)$ is a finite set V of nodes or vertices and a collection of edges, E. An edge is an unordered pair (u, v) of distinct vertices $u, v \in V(G)$. In general $V(G)$ and $E(G)$ will denote respectively the set of vertices and edges of the graph G.

A graph can be represented by a diagram in which a vertex is symbolized by a dot and an edge by a line segment connecting two dots.

Example 2.1. Figure 2.1 shows the graph $G = (V, E)$ where:

$$\begin{aligned} V &= \{a, b, c, d, e, f, g, h, i, j, k\} \\ E &= \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}\} \end{aligned}$$

The edges can also be denoted by the corresponding end vertices, for instance $e_6 = (d, e)$ or $e_{13} = (g, i)$.

The *degree of a vertex* $v \in V(G)$, denoted by $d(v)$ is the number of edges $E(G)$ incident to v. An *isolated vertex* is a vertex with degree zero.

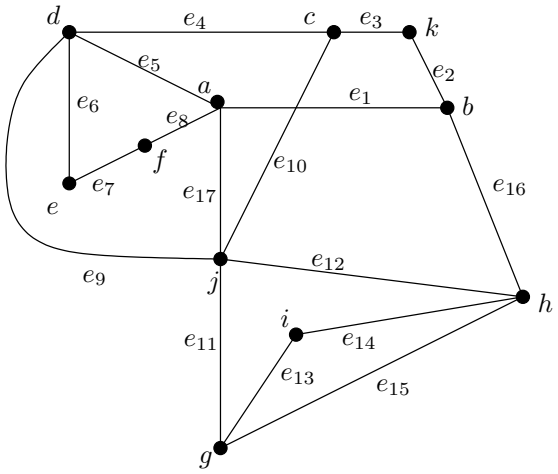


Figure 2.1: Graph example.

Example 2.2. In the graph G in Figure 2.1, for example, $d(a) = 4$, $d(e) = d(i) = d(k) = 2$.

Let $G = (V, E)$ be a graph. We say $G' = (V', E')$ is a *subgraph* of G if $V' \subset V$, $E' \subset E$ and an edge $(v_i, v_j) \in E'$ only if $v_i, v_j \in V'$.

Let $G = (V, E)$ be a graph and $V' \subset V$. Then the subgraph $G' = (V', E')$ is the *vertex-induced subgraph* of G if E' is a subset of E such that edge (v_i, v_j) is in E' if and only if v_i and v_j are in V'. That means that if $v_i, v_j \in V'$, then every edge in E having v_i and v_j as its end vertices should be in E'.

A *path* in a graph $G = (V, E)$ is a sequence of vertices $v_1, v_2, \ldots, v_n$ of $V(G)$ such that (v_i, v_{i+1}) is an edge in $E(G)$ for $1 \leq i < n$. A path is simple if all vertices on the path are distinct.

Example 2.3. See for example the simple path j, a, d, e, f in Figure 2.1.

In this work, a *circuit* is a simple path that contains no repeated vertices except v_1 and v_n which are the same. In what follows, we shall denote the set of vertices in a circuit as $C = \langle v_1, v_2, \ldots, v_{n-1}, v_n \rangle$ and we shall consider that vertices are circularly sorted. Whether vertices are sorted clockwise or counterclockwise is irrelevant.

Example 2.4. For example, $\langle g, i, h \rangle$ and $\langle h, i, g \rangle$ denote the same circuit in the graph in Figure 2.1.

2.2 Operations on Graphs

In this Section we introduce a few operations between graphs.

Consider two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. The *union* of G_1 and G_2, denoted as $G_1 \cup G_2$, is the graph $G = (V_1 \cup V_2, E_1 \cup E_2)$. The *intersection* of G_1 and G_2, denoted as $G_1 \cap G_2$, is the graph $G = (V_1 \cap V_2, E_1 \cap E_2)$.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. $G(V, E)$ is the *ring sum* of G_1 and G_2, denoted as $G_1 \oplus G_2$, when $V = (V_1 \cup V_2$, $E = (E_1 \cup E_2) - (E_1 \cap E_2))$, and all isolated vertices are dropped from V.

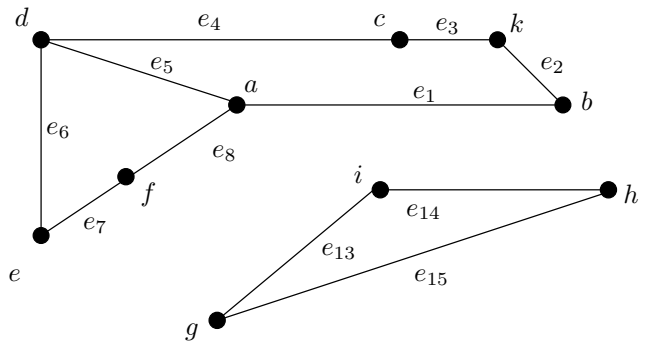


Figure 2.2: A disconnected graph.

Consider a graph $G = (V, E)$ and $v \in V$. Then *vertex removal operation*, denoted as $G' = G - v$, computes the vertex induced subgraph of G by $V - v$.

Consider $G = (V, E)$ a graph and $e \in E$. Then *edge removal operation*, denoted as $G' = G - e$, is the subgraph of G that results after removing from G the edge e_i. Note that the end vertices of e are not removed from G.

The removal of a set of vertices or edges from a graph G is defined as the removal of single vertices or edges in succession.

2.3 Connectivity of a Graph

Graph connectivity will play an important role in the algorithm proposed in this thesis. Given a graph G, the first point is to determine the connectivity of G.

A graph $G = (V, E)$ is *connected* if there exists a path between every pair of vertices in G, otherwise G is *disconnected*. The maximal connected subgraphs of a disconnected graph G are the *connected components* of G.

Example 2.5. For example, the graph G in Figure 2.2 is disconnected. Its two connected components are G_1 and G_2. Vertex sets are $V(G_1) = \{a, b, k, c, d, e, f\}$ and $V(G_2) = \{g, h, i\}$.

Let $G = (V, E)$ be a connected graph. A vertex $v \in V(G)$ is an *articulation vertex* if $V(G) - \{v\}$ is disconnected.

Theorem 2.1. *Let $G = (V, E)$ be a connected graph. If $v \in V(G)$ is an articulation vertex, there are two vertices $u, w \in V(G)$ with $u \neq v$ and $w \neq v$ such that v is on every path connecting vertices u and w.*

Proof. See the book of Thulasiraman and Swamy, [123]. □

A *non-separable* or *biconnected* graph $G = (V, E)$ has no articulation vertices, otherwise it is *separable*. A *biconnected component* of a connected graph G is a maximal biconnected subgraph of G. A connected graph can be decomposed into biconnected components. For any biconnected graph $G = (V, E)$, given a pair of vertices $u, v \in V(G)$, with $u \neq v$, there are, at least, two disjoint paths connecting u and v.

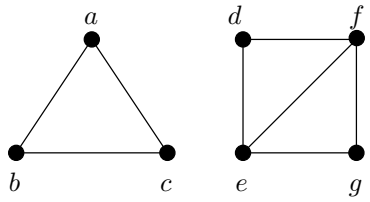


Figure 2.3: A 0-connected graph.

The *connectivity* of a graph $G = (V, E)$ is the minimum number k of vertices that must be removed to disconnect G. If the connectivity of G is k, we write $\kappa(G) = k$. For a disconnected or 0-connected graph G, $\kappa(G) = 0$. For a connected graph G, $\kappa(G) \geq 1$. A separable graph or 1-connected graph G has $\kappa(G) = 1$. A 2-connected G has $\kappa(G) = 2$. A biconnected graph has connectivity $\kappa(G) \geq 2$. In a similar way, a graph G with $\kappa(G) \geq 3$ is called *triconnected*. Biconnected graphs can be decomposed into triconnected components.

Example 2.6. Figure 2.3 shows a 0-connected graph and Figure 2.4 shows a 1-connected graph.

A star graph is a connected graph $G = (V, E)$ with one vertex $v \in V$ called *center* with degree greater than 1 and such that for each vertex $v_i \in V$ - $\{v\}$, the degree is 1.

2.4 Trees and Fundamental Circuits

In this Section we introduce trees and also point out the relationship between trees and circuits.

A graph is said to be *acyclic* if it has no circuits. A *tree* of a graph $G = (V, E)$ is a connected acyclic subgraph of G. A *spanning tree* T for a graph G is a tree that connects all the vertices in $V(G)$. The edges of a spanning tree T are called the *branches* of T. The edges $E(G)$ which are not in T are called the *chords*. If $|V(G)| = n$ and $|E(G)| = m$, the spanning tree T has $n - 1$ branches and $m - n + 1$ chords. A connected graph always contains a spanning tree.

The *co-spanning tree* T^* of a spanning tree T for a graph G is the subgraph of G whose vertices are those of G and whose edges are exactly those edges of G that are not in T.

Example 2.7. Figure 2.5 shows a graph G, a spanning tree T and the corresponding co-spanning tree T^*.

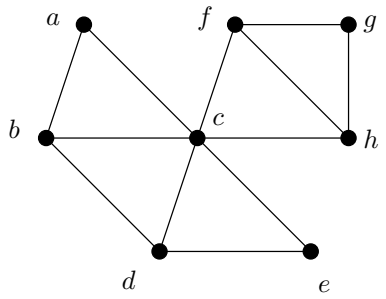


Figure 2.4: A 1-connected graph.

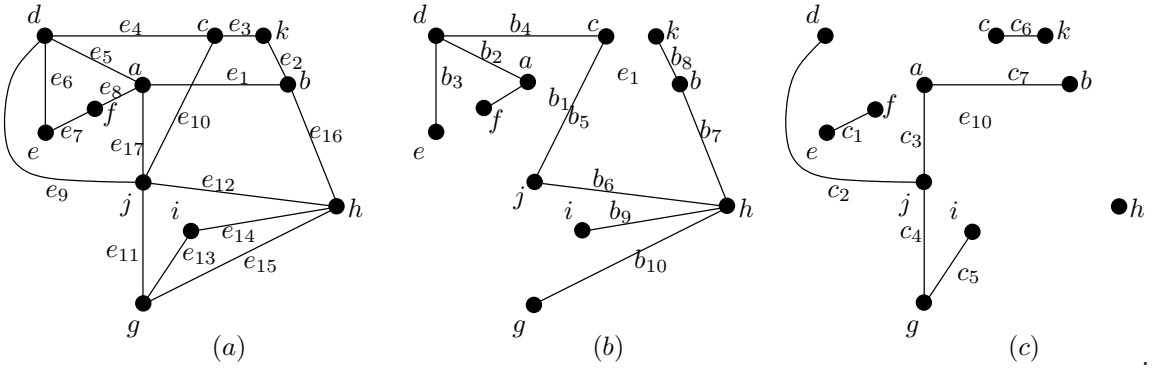


Figure 2.5: (a) A graph G, (b) A tree T of G and (c) A co-spanning tree T^* of T.

Theorem 2.2. *Let $G = (V, E)$ a graph with $|V| = n$ and $G' \subset G$. G' is a* spanning tree *of G if and only if G' is acyclic, connected, and has $n - 1$ edges.*

Proof. See Thulasiraman and Swamy, [123]. □

For a given spanning tree, a unique circuit can be obtained by adding to the spanning tree each of the chords. The set of $m - n + 1$ circuits obtained in this way is called the *fundamental system* relative to the spanning tree. A circuit in the fundamental system is called a *fundamental circuit*.

Theorem 2.3 (Fundamental Circuit Theorem)**.** *If $G = (V, E)$ is a graph and T a specified spanning tree, any circuit C in G can be expressed as the ring sum of some fundamental circuits of G with respect to T. Further, no fundamental circuit is the ring sum of any of the others.*

Proof. See Even, [28]. □

Example 2.8. Consider the graph G and the spanning tree T in Figure 2.5. The fundamental circuits are the following (See Figure 2.6),

$$\begin{aligned} C_1 &= \langle e, d, a, f \rangle \\ C_2 &= \langle d, c, j \rangle \\ C_3 &= \langle a, d, c, j \rangle \\ C_4 &= \langle j, h, g \rangle \\ C_5 &= \langle g, h, i \rangle \\ C_6 &= \langle c, j, h, b, k \rangle \\ C_7 &= \langle a, d, c, j, h, b \rangle \end{aligned}$$

Every fundamental circuit contains exactly one chord c_i that is not present in any other fundamental circuit respect to T. For example, C_1 contains the chord (e, f).

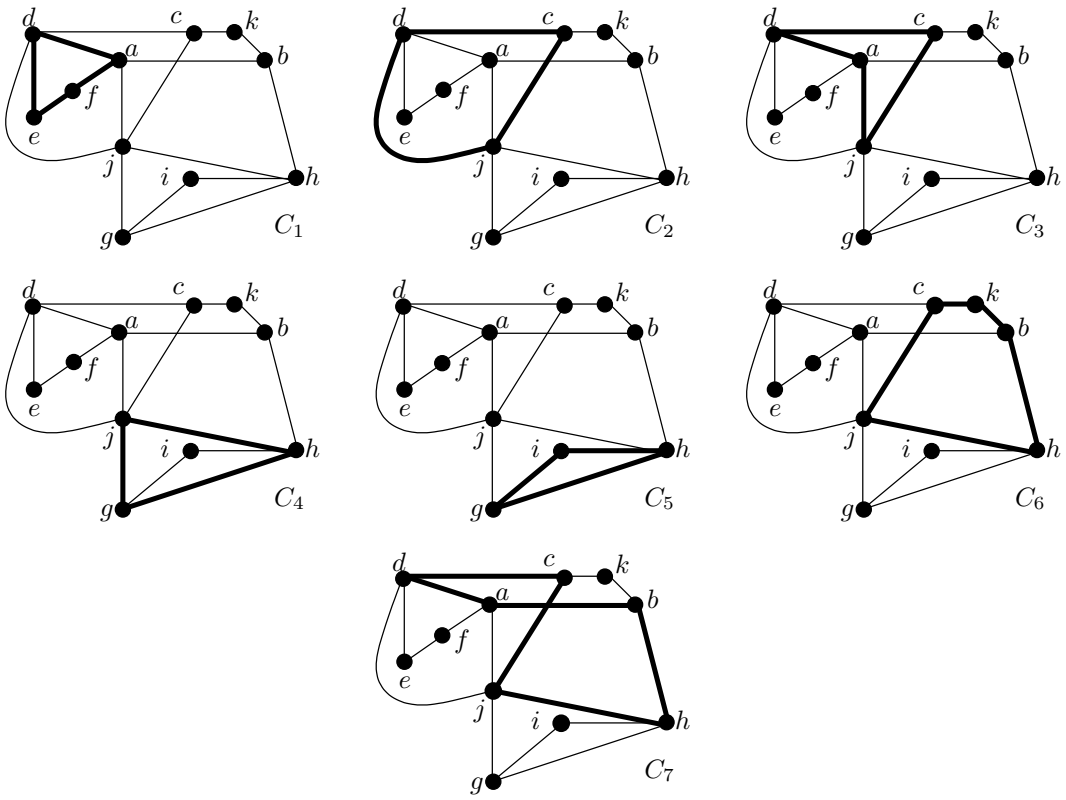


Figure 2.6: Fundamental circuits of a graph G.

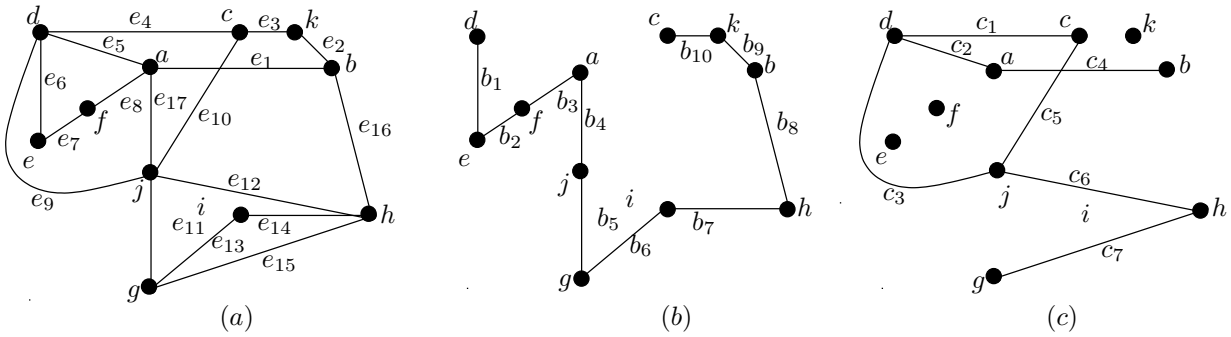


Figure 2.7: (a) G, (b) one of its DFS trees and (c) the co-spanning tree.

2.5 Depth-First Search Algorithm

The goal of this Section is to recall the Depth-First Search algorithm, in short DFS. The DFS algorithm is a technique of scanning a finite undirected graph. It allows to efficiently visit vertices we want to walk along the edges, from vertex to vertex and visit all vertices. Since the work of Hopcroft and Tarjan [51, 118] it is widely recognized as a powerful technique for solving a number of graph problems.

Assume we are given a finite connected graph $G = (V, E)$. Starting in one of the vertices we want to walk along the edges, from vertex to vertex and visit all the vertices. The start vertex v is called *the root of the DFS*. DFS terminates when the search returns to the root and all the vertices have been visited. When selecting the next vertex to explore, depth-first search selects a vertex that has never been explored and is connected by an edge to the most recently explored vertex. DFS partitions the edges of G into edges visited (branches) and edges not visited (chords). The branches form a spanning tree T of G. The chords form a co-spanning tree T^* of G.

Example 2.9. Figure 2.7 shows a graph $G = (V, E)$. Assuming the vertex d as the root, the DFS computes a spanning tree T. T^* is the corresponding co-spanning tree.

2.6 Planar Graphs and Embeddings

Planar graphs and embeddings are topics widely discussed in the literature. See, for example, [96, 97]. Since planar graphs play an important role in this work, here we introduce the main related concepts.

A graph $G = (V, E)$ is *planar* if there is a mapping from every vertex $v \in V(G)$ to a point in a two-dimensional space, and from every edge $e = (v_1, v_2) \subseteq E$ to a plane continuous curve such that the bounding points of each curve are the points mapped from the edge end vertices and all curves are disjoint except on their bounding points. The resulting projection is called a *plane graph* or *planar embedding*. In this work we consider planar graphs and embeddings, therefore we will use just the word *embedding* to refer to a planar embedding.

The complement with respect to the two-dimensional plane of an embedding is a collection of two-dimensional regions, from now on *faces*. Every face is homeomorphic to an open,

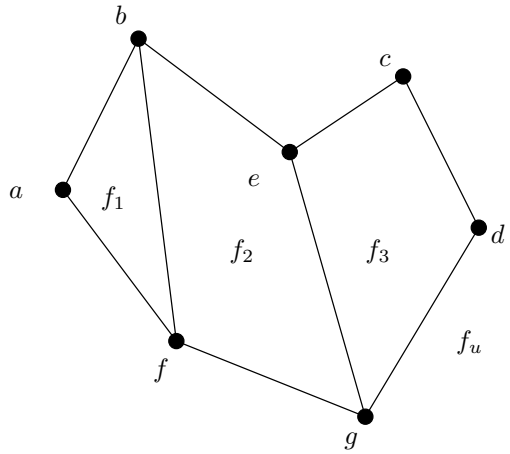


Figure 2.8: Planar embedding of a graph.

regularized disc. An embedding of a planar graph has an external or unbounded face. However, none of the faces of the embedding have particular properties. In general, the embedding of a planar graph is not unique. We let $F(G)$ denote the set of all faces of G.

Each face is bounded by a circuit of mapped vertices that unambiguously describe the face, that is, the *the boundary* of a face.

Theorem 2.4 (Euler's Formula)**.** *Let $G = (V, E)$ a connected planar graph with $F(G)$ faces. Then, $|V(G)| - |E(G)| + |F(G)| = 2$.*

Proof. See Even, [28], *p.*182. □

Example 2.10. Figure 2.8 depicts a planar embedding of a graph with four faces. Face $f_u = \langle a, b, c, d, e, f, g\rangle$, is the unbounded face and the other faces are $f_1 = \langle a, b, f\rangle$, $f_2 = \langle b, e, f, g\rangle$, and $f_3 = \langle c, d, e, g\rangle$. Note that $|V(G)| = 7$, $|E(G)| = 9$ and $|F(G)| = 4$ and the Euler's formula is accomplished.

2.7 Components and Bridges

Following Even, [28], and Voss *et al.*, [129], in this section we recall the concepts of components and bridges induced in a graph $G = (V, E)$ by a subset of vertices $S \subset V(G)$. Then we explain how a non-separable graph can be decomposed into one circuit and a set of bridges. This is achieved by first showing how an arbitrary set of vertices induces a graph decomposition. The result of this decomposition is a set of graph components. If a number of conditions hold in the decomposition, then the resulting components are bridges.

Let $G = (V, E)$ be a non-separable graph and let $S \subseteq V$. Consider the partition of the set $V - S$ into classes such that two vertices are in the same class if and only if there is a path connecting them which does not use any vertex of S. Let $\equiv$ denote this relationship and assume that $V(G) - S/\equiv = \{K_1, \ldots, K_m\}$. Each class K_i defines a *non-singular component* $H = (V', E')$ of G, such that

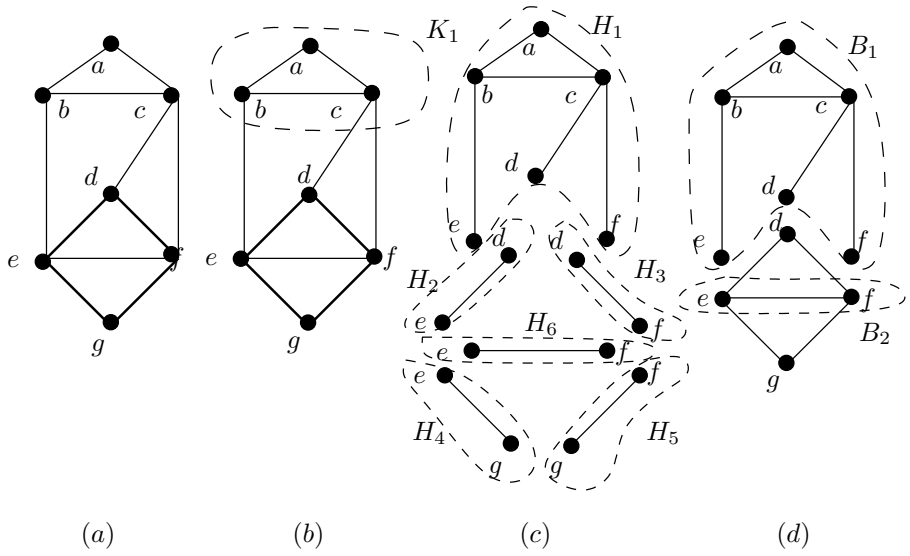


Figure 2.9: (a) Graph, (b) Classes, (c) Components and (d) Bridges.

- $V' \supset K_i$.
- V' includes all the vertices of C which are connected by an edge to a vertex of K_i.
- $E' \subseteq E$ contains all those edges of $E(G)$ which have at least one end vertex in K_i.

An edge $(u, v) \in E(G)$, where both $u, v \in S$ defines a *singular component* denoted $H = (\{u, v\}, \{(u, v)\})$. Therefore a set of vertices $S \subset V(G)$ induces a decomposition of G into a set of singular and non-singular components $\{H_1, \ldots, H_n\}$. Note that two components share no edges, and the only vertices they can share are vertices of S.

Now we define the set of bridges of a graph G with respect to a circuit C.

Let C be a circuit of the graph G and consider the set of vertices $V(C)$. Let H be the set of components induced in G induced by $V(C)$. A *bridge*, B_i, is a *component* $H_i \in H$ whose edges do not belong to C. A bridge is said to be singular if the corresponding component is singular, otherwise it is non-singular. The vertices of a bridge $v \in V(H)$ such that $v \in V(C)$ are called *attachments* and will be denoted as $at(H)$. In what follows, vertices in bridge attachments are considered sorted according to the circular order defined in the vertices of the circuit which originated the bridges.

Let $C = \langle v_1, v_2, \ldots, v_n \rangle$ be the fundamental circuit under consideration and let B be a bridge with attachments $at(B) = \{a_1, a_2, \ldots, a_m\} \subset V(C)$ where vertices are sorted according to the circular order in C. We define the *span* of a bridge B over a circuit C by $sp(B) = \{v \in V(C) | a_1 < v < a_m\}$. By *span length* we mean the number of vertices in the span plus two.

Example 2.11. Consider the graph $G = (V, E)$ given in Figure 2.9(a) and let $S = [d, e, g, f] \in V$ shown in bold. Since $V - S = \{a, b, c\}$, there is just one induced class $K_1 = \{a, b, c\}$ shown

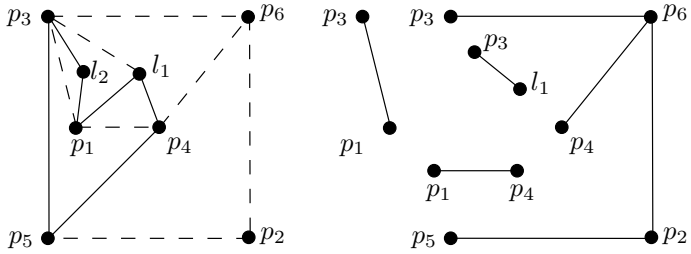


Figure 2.10: (a) Graph. (b) Bridges.

in Figure 2.9(b). The components, shown in Figure 2.9(c), are

$$\begin{aligned} H_1 &= (\{a,b,c,e,d,f\},\{(a,b),(a,c),(b,c),(b,e),(c,d),(c,f)\}) \\ H_2 &= (\{d,e\},\{(d,e)\}) \\ H_3 &= (\{d,f\},\{(d,f)\}) \\ H_4 &= (\{e,g\},\{(e,g)\}) \\ H_5 &= (\{g,f\},\{(g,f)\}) \\ H_6 &= (\{e,f\},\{(e,f)\}) \end{aligned}$$

H_1 is a *non-singular component*, and H_i, $2 \leq i \leq 6$, are singular components.

Components H_1 and H_6 are bridges, shown in Figure 2.9(d) respectively as B_1 and B_2. Attachments of B_1 are $\{e,d,f\}$ and attachment of B_2 are $\{e,f\}$. B_2 is a singular bridge. For example, for the bridge B_1, we have that the bridge span is $sp(B_1) = (d)$ with a span length equal to three.

Example 2.12. For example consider now case study graph G and circuit C in bold shown in Figure 2.10(a). The bridges are shown in Figure 2.10(b).

Theorem 2.5. *Let B be a bridge with attachments a_1, a_2 and a_3. There exists a vertex $v \in B$, that is not an attachment, for which there are three vertex disjoint paths in B: $v - a_1$, $v - a_2$ and $v - a_3$.*

Proof. See Even, [28]. □

Example 2.13. Let us consider the graph in the Example 2.11. The attachments of bridge B_1 are $\{e,d,f\}$. Vertex $a \in B_1$ is not an attachment and we can find the paths $\{a,b,e\}$, $\{a,c,d\}$ and $\{a,c,f\}$.

2.8 Planar Graphs and Bridges

This section is devoted to the relationships between planarity and bridges. First, we introduce the concept of bridge interlacement. Then we consider the planar embedding of bridges.

Following Even, [28] and Bondy and Murty, [9] we define the concept of bridges interlacement.

Let $G = (V,E)$ be a non-separable graph and C be a circuit in G. Let $B_1, \ldots, B_k$ be the bridges of G with respect to C. We say that B_i, B_j interlace if at least one of the following condition holds:

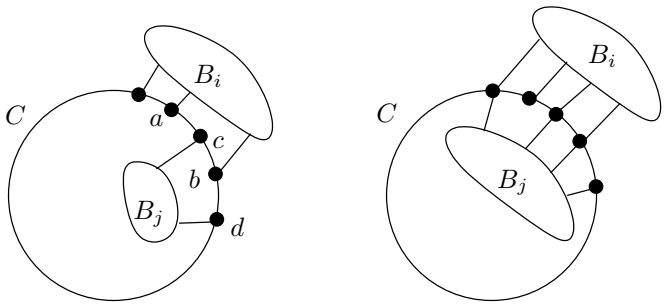


Figure 2.11: (a) Bridges interlacement case 1. (b) Bridges case 2.

1. There are four distinct vertices a, b, c, d in increasing order in C such that either a, c belong to $at(B_i)$ and b, d belong to $at(B_j)$ or a, c belong to $at(B_j)$ and b, d belong to $at(B_i)$; or

2. $at(B_i)$ and $at(B_j)$ share three distinct vertices.

Figure 2.11 illustrates these two conditions.

Example 2.14. Consider the graph G in Figure 2.12. Let $C = \langle b, k, c, d, e, f, a\rangle$ be the circuit shown in bold. The bridges of the graph G are B_1, B_2 and B_3, with $at(B_1) = \{d, c, b\}$, $at(B_2) = \{d, a, c, b\}$ and $at(B_3) = \{d, a\}$. B_1 and B_2 interlace because they have three common attachments.

Next follows the relation between embeddings and non-interlacing bridges. For each bridge B_i, consider the subgraph $C \cup B_i \subseteq G$. If any of these graphs $C \cup B_i$ of G is not planar, then clearly G is not planar. Now, assume all these subgraphs are planar. In every planar embedding of G, C outlines a contour which divides the plane into two disjoint parts, one is bounded (inside) and the other is unbounded (outside). Two bridges that do not interlace can be embedded in any side of the circuit C. However, if two bridges interlace they cannot be embedded on the same side of C. Thus, in every planar embedding of G, C partitions the interlacing bridges into two sets: those which are drawn inside C and those which are drawn outside. No two bridges in the same set interlace. Notice that choosing the side where the bridges are embedded does not matter.

Theorem 2.6. *If B_i, $1 \leq i \leq m$, is the set of bridges of a non-separable graph G with respect to a simple circuit C and the following two conditions are satisfied:*

1. For every i, $C \cup B_i$ is planar,

2. No two bridges interlace,

then $C \cup (\bigcup_{1 \leq i \leq m} B_i)$, has a planar embedding in which all these bridges are inside (or outside) of C.

Proof. See Even, [28]. □

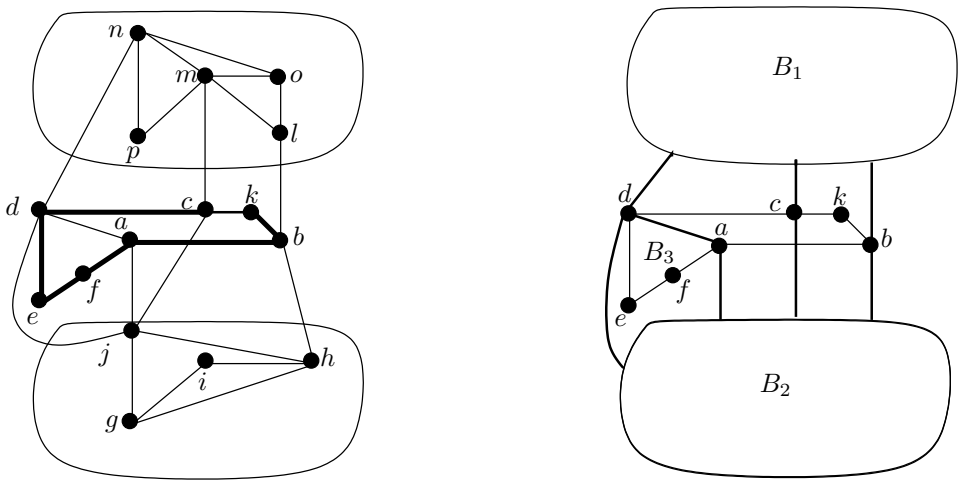


Figure 2.12: Interlacing bridges B_1 and B_2.

2.9 Chapter Summary

In this Chapter we have presented the main concepts and properties of graph theory that are the base of our $DR-planner$ algorithm. The concepts of k-connectivity of a graph, tree, depth-first search procedure, planar graphs, bridges, bridges interlacement and graph embedding will be widely used in Chapter 4.

CHAPTER 3

Geometric Constraint Solving Background

> It is better to know some of the questions than all of the answers.
>
> *(James Thurber)*

In this Chapter we start with an overview of the basic concepts related to geometric constraint problems. Then, we survey the current state of the art in geometric constraint solving.

Most of the work of this chapter has been previously published in references [59, 63, 119].

3.1 Geometric Constraint Problem

Following the works of Joan-Arinyo *et al.*, [61, 63, 68], Hoffmann *et al.*, [49] and Sitharam *et al.*, [40], we describe in detail the basic concepts related to geometric constraint problems.

3.1.1 The General Geometric Constraint Problem

The *general geometric constraint problem* consists of a set of geometric elements, a set of geometric constraints defined between them and a set of parameters, i.e. the symbolic values of the geometric constraints.

We consider geometric constraint problems in the Euclidean plane consisting of a finite set of geometric objects and a finite set of constraints defined between them. The geometric objects are drawn from a fixed set of types such as points, straight lines, circles and arcs of circle. The constraints include topological constraints such as tangency, incidence and perpendicularity, and metric constraints such as point-point distance, perpendicular point-straight line distance and line-line angle.

A geometric constraint problem can be characterized by a tuple (E, O, X, C) where

- E is the geometric space where the problem is embedded. E is usually Euclidean.
- O is the set of specific geometric objects which define the problem. They are chosen from a fixed repertoire including points, lines, circles and the like.

- X is a possibly empty set of variables whose values must be determined. In general, variables represent quantities with geometric meaning: distances, angles and so on. When the quantities are without a geometric meaning, for example, when they quantify technological aspects or functional capabilities, those variables are called *external*.

- C is the set of constraints. Constraints can be geometric or equational. Geometric constraints are relationships between geometric elements chosen from a predefined set, e.g., distance, angle, tangency, etc. The relationship (the distance, the angle,...) is represented by a symbolic tag. If the tag represents a fixed value, that will be known in advance, then the constraint is called *valuated*. If the tag is a variable in X, then the constraint is called *symbolic*. Equational constraints are general equations defined on the set of variables X and tags T. The set of equational constraints can be empty.

Example 3.1. Figure 3.1 depicts an example of a general geometric constraint problem in $2D$ which components are the following:

- The set of geometric elements, $O = \{A, B, C, D, L_{AB}, L_{AC}, L_{BC}\}$.

- The set of tags in the constraints, $P = \{d, h, \alpha\}$.

- The set of geometric variables, $V_1 = \{x, y\}$.

- The set of external variables, $V_2 = \{v\}$.

- The set of geometric constraints with fixed tags, $C_1 = \{dpp(A, B) = d, dpl(C, L_{AB}) = h, \angle(L_{AB}, L_{BC}) = \alpha\}$.

- The set of geometric constraints with variable tags, that is, symbolic constraints, $C_2 = \{dpp(A, C) = x, dpp(C, D) = y\}$.

- The set of equational constraints, $C_3 = \{y = x \cdot v, v = 0.5cos(\alpha)\}$.

with $X = V_1 \cup V_2$ and $C = C_1 \cup C_2 \cup C_3$.

The meaning of the basic construction names is the usual: point defined by its coordinates, straight line given by an ordered pair of points. For example, L_{AB} defines a line between two points A and B. Predicate names are self explanatory. The predicate $dpp(A, B) = d$ specifies a point-point distance, $dpl(C, L_{AB}) = h$ defines the signed perpendicular distance from a point to a straight line and, $\angle(L_{AB}, L_{BC}) = \alpha$ denotes the angle between two straight lines.

Solving a given a geometric constraint problem (E, O, X, C) entails two questions:

1. Can be the geometric elements in O placed in E in such a way that the constraints in C are satisfied? If the answer is positive, then

2. Given an assignment of values to the symbolic constraints and the external variables, is there an actual placement of the elements in O that satisfies all the constraints in C?

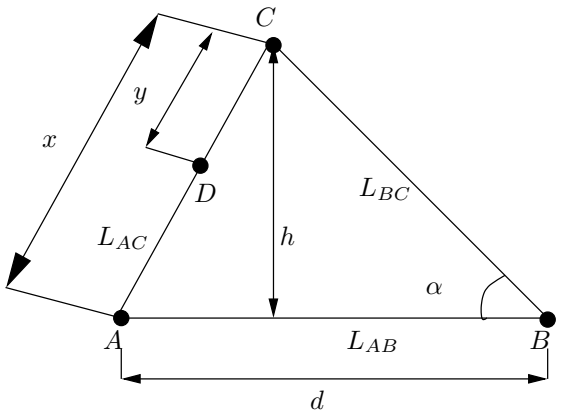


Figure 3.1: A general geometric constraint solving problem in 2D, where $y = x \cdot v$ and $v = 0.5cos(\alpha)$.

3.1.2 The Basic Geometric Constraint Problem

A basic geometric constraint problem only considers geometric elements and constraints whose tags are assigned a value. It excludes external variables, constraints whose tags must be computed, and equational constraints. Some constraints are assigned a symbolic value; the set of those symbolic values is named the set of parameters. We denote a GCP as a tuple (O, C, P) where O is the set of geometric elements, C the set of constraints and P the set of parameters.

Constraints in C are predicates defined over $O \cup P$. The geometric elements of O belongs to a catalog of available geometric elements $\mathcal{O}$. Usually $\mathcal{O}$ contains point, lines, circle, etc. In a similar way the constraints in C are chosen from a catalog of constraints, $\mathcal{C}$, which contains constraints like point-point distance, perpendicular point-line distance, or angle between straight lines.

3.1.3 Rigidity Characterization

We are interested in describing shapes which are invariant under rigid transformations of translation and rotations. We will say that these shapes are *rigid*. Different kinds of rigidity have been defined, such as minimal or global rigidity. See examples of different definition for rigidity in the works of Servatius [107], Jackson [54], Laman [81], Hoffmann *et al.*, [46], Whiteley, [133], Thierry, [122], Sitharam *et al.*, [40], Michelucci *et al.*, [91] and Zhang, [136].

Intuitively, solving a geometric constraint problem can be understood as follows. Let $O \in \mathcal{O}$ be a set of geometric elements. Then to place O means to obtain a vector $\vec{r} \in \mathbb{R}^u$ that allows to embed each element of O into the Euclidean plane. The vector $\vec{r}$ is named a *placement* of O. A placement $\vec{r}$ allows to map a set of abstract geometric elements O to a set of actual geometric elements embedded in the plane noted as $\vec{r}.O$.

Example 3.2. Say, for instance, that a set of geometric elements O is composed by $O = \{l_1, p_1, p_2\}$ being p_i a point and l_i a line as usual. Assume that a line l_i is represented by an equation $l_i : y = a_i x + b_i$ and a point by the pair of coordinates $p_i = (x_i, y_y)$. Note that to place $l_1 \in O$ means to determine a pair of real values a_1 and b_1 that map l_1 to the actual line $y = a_1 x + b_1$ in the plane.

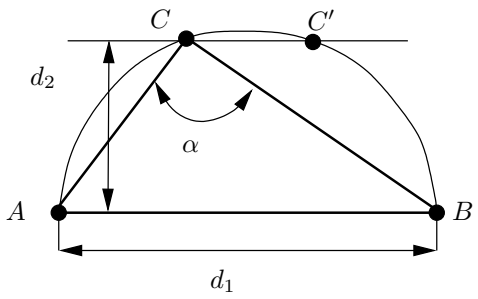


Figure 3.2: An structurally well-constrained problem.

Therefore a placement $\vec{r}$ of O is a vector in $\mathbb{R}^6$ that allows to place every element of O in the plane. If $\vec{r} = (1, 2, 3, 4, 5, 6)$ for instance, then $\vec{r}.O = \{l_1 : y = 1x + 2, p_1 : (3, 4), p_2 : (5, 6)\}$.

Let $P = \{p_1, \ldots, p_n\}$ be a parameters set. An *assignment* to P is a vector $\vec{t} \in \mathbb{R}^w$ where $w = |P|$. Note that an assignment $\vec{t}$ maps every formal parameter of P to an actual value in $\mathbb{R}$. Following placements, an assignment $\vec{t}$ applied to a set of constraints C replaces each parameter p_i in C by its actual value. The resulting constraint set is noted as $\vec{t}.C$.

An abstract GCP $S = (O, C, P)$ can be transformed into an actual one by applying an assignment. The result of applying an assignment, say $\vec{t}$, to S is the actual GCP $\vec{t}.S = (O, \vec{t}.C, P)$. In these conditions, we can precisely define the concept of solving a geometric constraint problem as follows

Definition 3.1. Let $\vec{t}.S = (O, \vec{t}.C, P)$ be an actual GCP. We say that a placement $\vec{r}$ is a solution of $\vec{t}.S$ if and only if the actual geometric elements of $\vec{r}.O$ satisfy all the actual constraints in $\vec{t}.C$. In general, for a given assignment $\vec{t}$ there are a number of placements $\vec{r}$ which are solution of $\vec{t}.S$.

However, Definition 3.1 hides many corner cases that should be considered. In fact it is possible for a given GCP and a particular assignment to get an infinite number of solutions. Indeed, there are GCP such that for most of the assignments there is an infinite number of solutions. We can also find GCP such that for most of the assignments there is no solution.

Rigidity theory uses the concept of generically well constrained to denote an abstract problem which is well defined. In some sense an abstract problem is generically well constrained if it is well constrained for all the parameter assignments except for those degenerate ones. Thus, abstract geometric constraint problems can be classified according to the following criteria:

1. A GCP is structurally *well constrained* when for all of the assignments but a finite number of them there is a finite non-empty number of solutions.

   Example 3.3. Following [59], in Figure 3.2, the height of the triangle, d_2, an edge length, d_1 and the angle α constraints define a triangle with a nonempty, finite number of solutions. Specifically, the figure shows two possible triangles including respectively vertices C and C'.

2. A GCP is structurally *under-constrained* if for all of the assignments except a finite number of them the number of solutions is infinite.

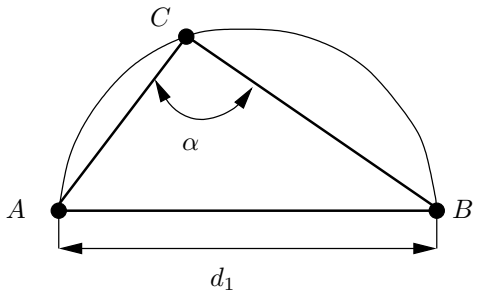


Figure 3.3: An structurally under-constrained problem.

Example 3.4. For instance consider the Figure 3.3, [59]. As long as vertex C is on the arc of circle, the triangle is well defined. Adding one more constraint results in an structurally well-constrained triangle.

3. A GCP is structurally *over-constrained* if for all of the assignments but a finite number of them the problem has no solution. Generally the removal of one or more constraints results in a well constrained problem. See the works of Hoffman *et al.*, [50], and Jermann *et al.*, [56].

   Example 3.5. See Figure 3.4, [59]. In this case, for most values of d_3, the problem has no solution.

In the rest of the manuscript we will use the term well constrained and under-, over-constrained to refer to a structurally well (under-, over-) constrained geometric constraint problem.

For our interest, we capture the notion of structural rigidity following Laman's Theorem, [81].

Theorem 3.1. *Let $G = (V, E)$ be a geometric constraint graph corresponding to a graph representation of a geometric constraint problem 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 *if and only if*

1. $|E| = 2|V| - 3$ *and*

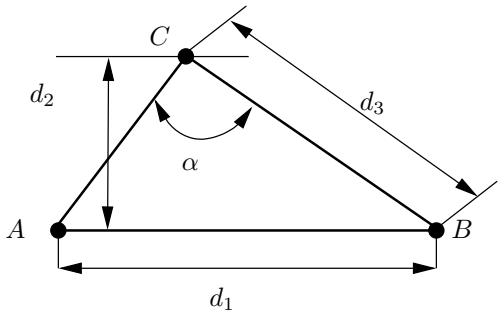


Figure 3.4: An structurally over-constrained problem.

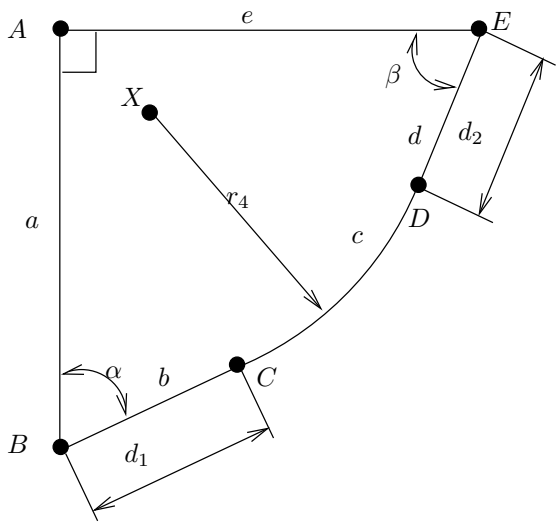


Figure 3.5: A geometric constraint problem.

2. *for every set of geometric elements* $V' \subseteq V$*, the induced subgraph* $G' = (V', E')$ *fulfils that* $|E'| \leq 2|V'| - 3$

3.2 Representation of Geometric Constraint Problems

As has been described above, geometric constraint problems include geometric elements and constraints on them. Several techniques to represent the geometric constraint problems have been developed. These methods can be roughly classified into equational, logic-based and graph-based.

Among the various forms see, eg, Hoffmann *et al.*, [6, 49], Joan-Arinyo *et al.*, [61], Soto-Riera, [113], Vila-Marta, [128] and Jermann *et al.*, [58]. Most of them use a graph-based approach representation. We summarize the relevant representation methods and we illustrate the representation of a geometric problem using the example of Figure 3.5, [12].

3.2.1 Equational Representation

Equational techniques translate the problem to a system of equations. For instance, a $2D$ point can be modeled by its Cartesian coordinates (x, y) and a point-point distance can be modeled by the classical equation $(x_1 - x_2)^2 + (y_1 - y_2)^2 = d^2$.

Example 3.6. For example, equations for the problem in the case of Figure 3.5 are given in Figure 3.6. Equations (3.1*a*) and (3.1*b*) captures the idea that lines are normalized, a common hypothesis used in the field of geometric constraint solving to facilitates both reasoning and geometric calculations.

$$a_x B_x + a_y B_y + a_r = 0 \tag{3.1a}$$
$$a_x^2 + a_y^2 = 1 \tag{3.1b}$$
$$b_x B_x + b_y B_y + b_r = 0 \tag{3.1c}$$
$$b_x^2 + b_y^2 = 1 \tag{3.1d}$$
$$(B_x - C_x)^2 + (B_y - C_y)^2 = d_1^2 \tag{3.1e}$$
$$(C_x - X_x)^2 + (C_y - X_y)^2 = r^2 \tag{3.1f}$$
$$b_x X_x + b_y X_y + b_r = r \tag{3.1g}$$
$$a_x = -e_y \tag{3.1h}$$
$$a_y = e_x \tag{3.1i}$$

Figure 3.6: Equations for the problem in Figure 3.5.

3.2.2 Logic-Based Representation

In the logic-based representation, the geometric problem is translated into a set of assertions, axioms and first order predicates which characterize the constraints and the geometric objects. This approach is especially interesting when the problem includes non-geometric information, in particular, equations, Soto-Riera, [113].

Example 3.7. Figure 3.7 shows part of a logic-based representation for the problem in Figure 3.5. Notice, for example, that the geometric elements are represented as predicates *Points* or *Lines*, and the axioms such that $distancePP(B, C, d_1)$ refers to the distance d_1 between points B and C.

Geoms
- Points(A, B, C, D, E, X)
- Lines(a, b, d, e)
- Circles(c, r)

Constraints
- on(A, a), on(A, e),...
- tangent(b, c),...
- distancePP(B, C, d_1),...
- center(c, X)
- ...

Figure 3.7: Logic-based representation for the problem in Figure 3.5.

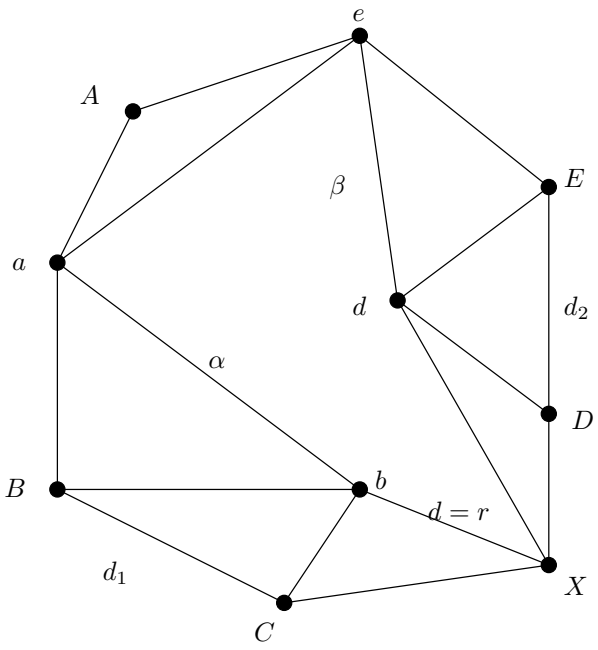


Figure 3.8: Graph representation of the problem in Figure 3.5.

3.2.3 Graph-Based Representation

In this technique the problem is translated into a graph where nodes represent geometric elements and edges represent geometric constraints between them. Geometric constraint graphs are simple graphs.

Example 3.8. Figure 3.8 shows the graph representation corresponding to the geometric constraints problem in Figure 3.5. Notice that points A and B and straight segments e and a correspond to the vertices of the graph. Edges of the graph correspond to constraints between geometric elements. For example, the distance between points B and C corresponds to graph edge labeled d_1.

Example 3.9. Following [49, 68], Figure 3.9 shows a geometric constraint problem and the associated constraint graph.

Examples of Graph Representation

Mechanisms in the Euclidean plane are interesting examples of geometric constraint problems which can be abstracted as constraint graphs.

Example 3.10. Figure 3.10 shows a frontal view of the mechanism used for opening and closing the doors of an elevator, [76]. The mechanism is a ten-bar linkage, consisting of two interconnected six-bar mechanisms which share one link l_1 and the ground. Only one of the six-bar mechanisms is described here.

The passenger compartment is shown as a dotted rectangle. Link l_1 is a flywheel which in the real mechanism is driven by a motor; since only two points on the flywheel are of interest,

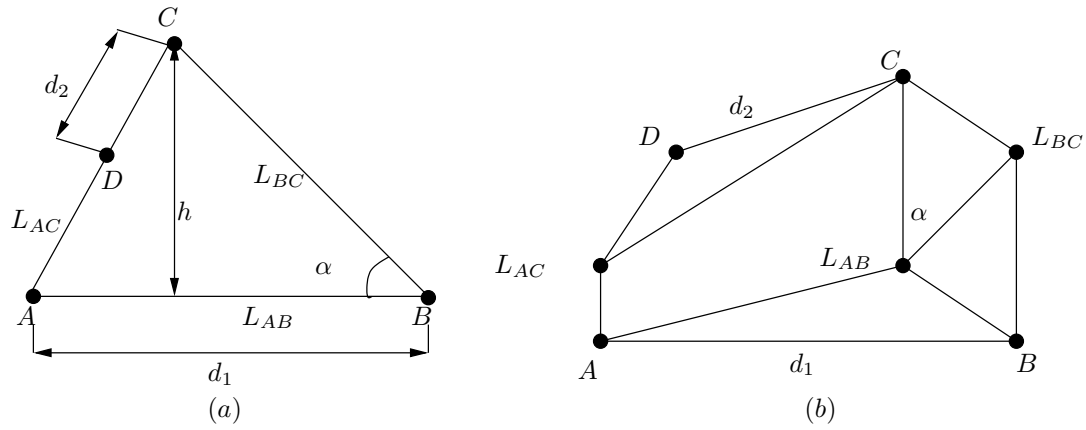


Figure 3.9: (a) A geometric constraint problem and (b) the associated constraint graph.

it is represented here as a line. As link l_1 rotates, it moves the push-rod link l_2, causing link l_3 to rotate about its grounded joint. Link l_4 is connected to link l_5, the elevator door, which is drawn as a black rectangle to aid visualization of the mechanism. The door slides on a prismatic joint. Link l_4 is necessary because the end of link l_3 moves in a circular path and link l_5 moves in a straight line.

The six-bar linkage for the other door operates in the same way. Note that the mechanism is not quite symmetric; since the two push-rods are connected to diametrically opposite points on the flywheel, the motion of each door is slightly different. The dimensions of the linkage are chosen to minimize the difference between door motions, and to make the rate at which the doors move (which varies over the motion) aesthetically pleasing to the passengers.

Figure 3.11 shows the elevator door mechanism with the associated geometric constraints. The set of geometric constraints are those listed in Figure 3.12 which includes point-point distance constraint, $dpp()$, coincidence, $on()$, point-line distance $dpl()$ and angle between two straight segments, $angle()$.

The geometric constraint problem can be represented by means of a *geometric constraint graph* $G = (V, E)$, where the nodes in V are geometric elements with two degrees of freedom and the edges in E are geometric constraints such that each of them cancels one degree of freedom.

Figure 3.13 shows the geometric constraint graph corresponding to the described elevator door mechanism.

Example 3.11. Figure 3.14 (b) shows a crank-connecting rod constraint mechanism that turns an alternative movement in a rotational motion, [32, 49]. This 2D mechanism can be abstracted by the geometric constraint problem in Figure 3.14 (a). The linear motion of the point p_5 along the straight line l_1 is transformed into a circular motion of the point p_4 along the circle with center on point p_3 and radius d_3.

The problem is defined by the tuple $CR = (E, C, P)$ where the set E contains the following geometric elements:

- Five points, p_i for $1 \leq i \leq 5$,

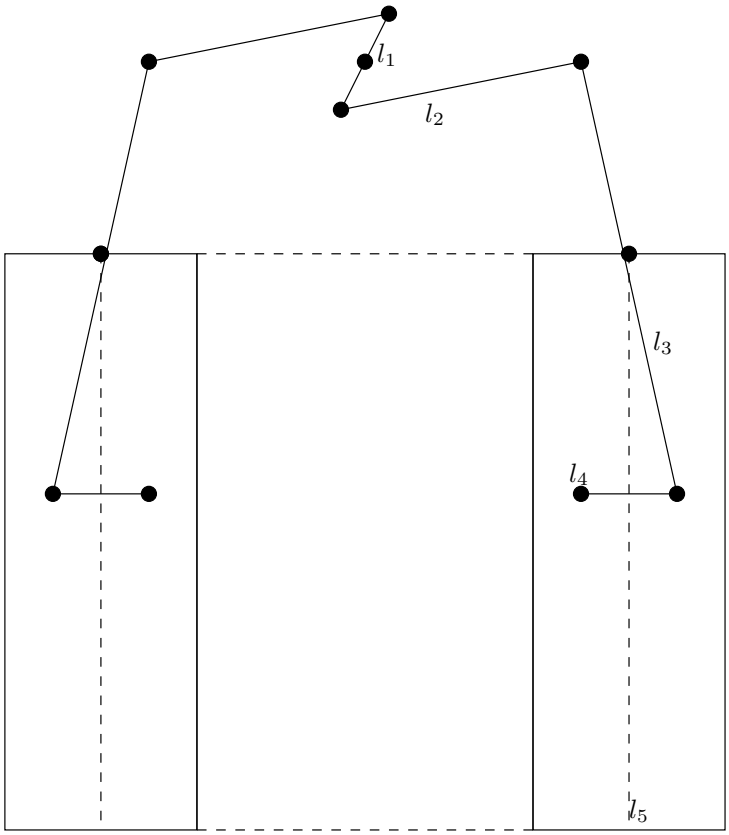


Figure 3.10: A frontal view of an elevator door actuator mechanism.

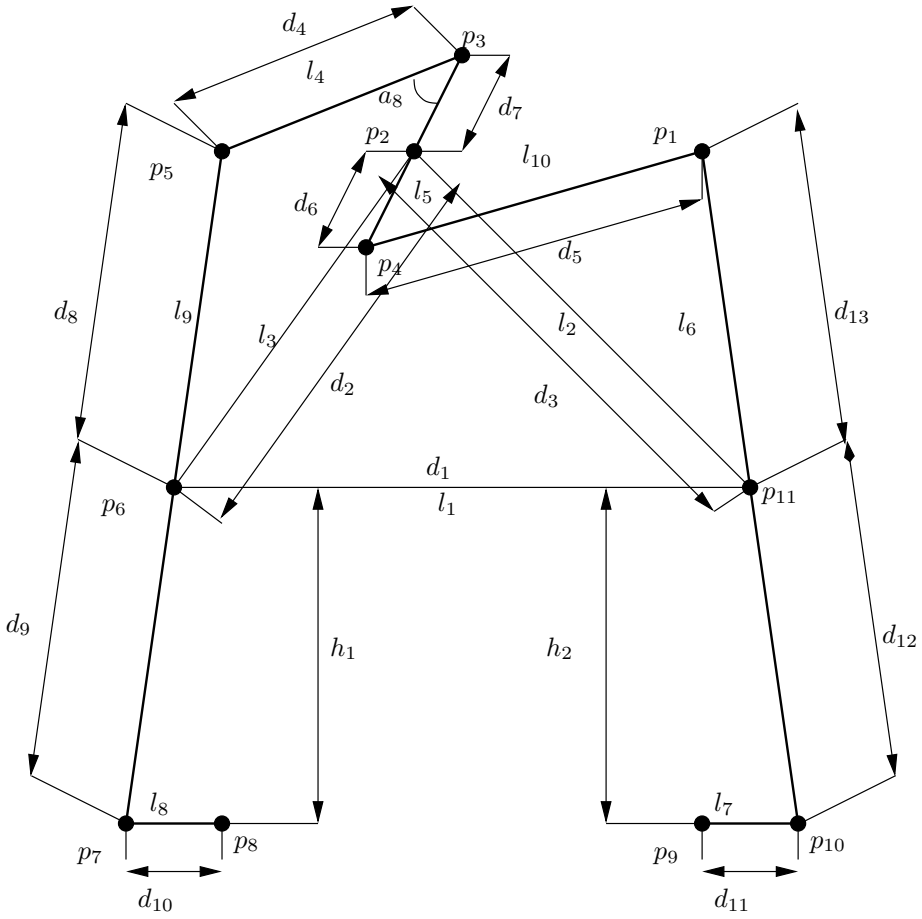


Figure 3.11: An elevator door actuator mechanism with its associated geometric constraints.

1.	dpp(p_1, p_4, d_5)	21.	on(p_{11}, l_6)
2.	dpp(p_1, p_{11}, d_{13})	22.	on(p_6, l_9)
3.	dpp(p_2, p_6, d_2)	23.	on(p_6, l_1)
4.	dpp(p_2, p_{11}, d_3)	24.	on(p_6, l_3)
5.	dpp(p_2, p_4, d_6)	25.	on(p_7, l_9)
6.	dpp(p_2, p_3, d_7)	26.	on(p_7, l_8)
7.	dpp(p_3, p_5, d_4)	27.	on(p_8, l_8)
8.	dpp(p_5, p_6, d_8)	28.	on(p_4, l_{10})
9.	dpp(p_6, p_7, d_9)	29.	on(p_4, l_5)
10.	dpp(p_6, p_{11}, d_1)	30.	on(p_2, l_5)
11.	dpp(p_7, p_8, d_{10})	31.	on(p_2, l_2)
12.	dpp(p_9, p_{10}, d_{11})	32.	on(p_2, l_3)
13.	dpp(p_{11}, p_{10}, d_{12})	33.	on(p_3, l_5)
14.	on(p_1,l_{10})	34.	on(p_3, l_4)
15.	on(p_1, l_6)	35.	on(p_5, l_4)
16.	on(p_{10}, l_6)	36.	on(p_5, l_9)
17.	on(p_{10}, l_7)	37.	dpl(p_9, l_1, h_1)
18.	on(p_9, l_7)	38.	dpl(p_8, l_1, h_2)
19.	on(p_{11}, l_1)	39.	angle(l_4, l_5, a_8)
20.	on(p_{11}, l_2)		

Figure 3.12: Geometric constraints for the elevator door mechanism of Figure 3.11.

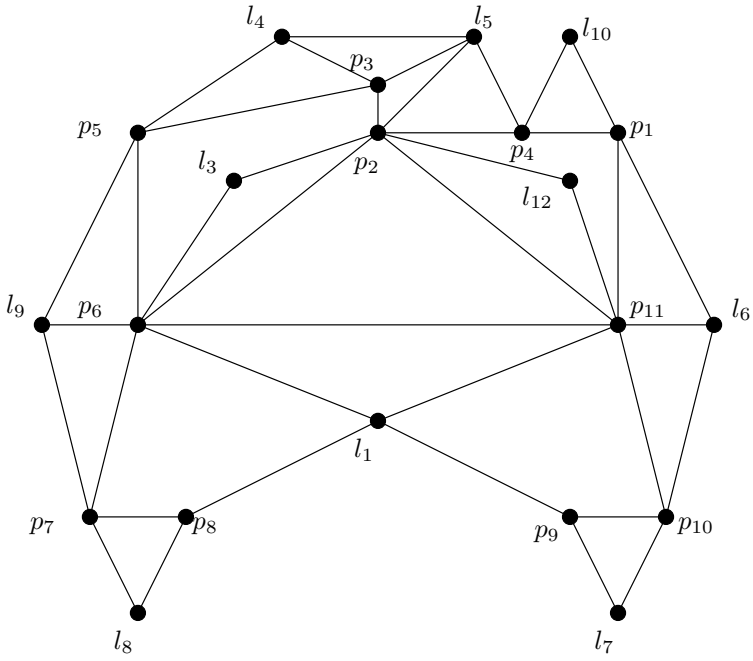


Figure 3.13: The Geometric Constraint Graph for the elevator door mechanism in Figure 3.11.

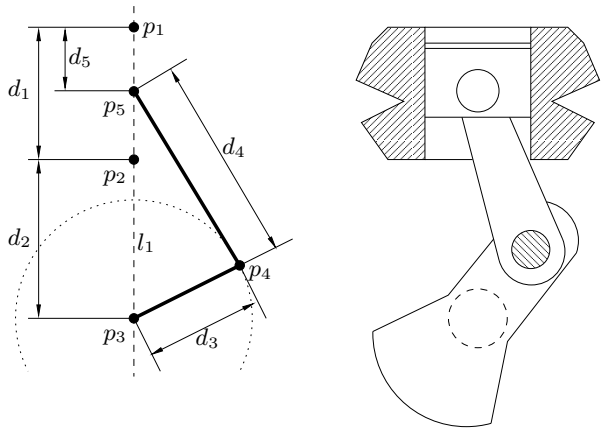


Figure 3.14: Crank and connecting rod. (a) Abstract geometric problem. (b) Actual mechanism.

- A straight line l_1.

The set of constraints C contains the following elements:

$$\begin{array}{ll} dpp(p_1, p_2, d_1) & on(p_1, l_1) \\ dpp(p_2, p_3, d_2) & on(p_2, l_1) \\ dpp(p_3, p_4, d_3) & on(p_3, l_1) \\ dpp(p_4, p_5, d_4) & on(p_5, l_1) \\ dpp(p_1, p_5, d_5) & \end{array}$$

Where $dpp(p, q, d)$ is the predicate asserting that the distance between point p and q is d, and $on(p, l)$ asserts that point p is on line l. Finally the set of parameters P is $P = \{d_1, d_2, d_3, d_4, d_5\}$.

Figure 3.15 shows the geometric constraint graph for the crank-connecting rod problem in Figure 3.14(a).

3.3 Resolution Methods in GCS: An overview

In this section we review basic techniques that are available for solving $2D$ geometric constraint problems. A good survey of such techniques can be found in Joan-Arinyo and Hoffmann., [49], Hoffmann *et al*, [6, 36], Joan-Arinyo *et al.*, [19, 135], Sitharam, [109], Thierry, [120, 121] and references therein.

The main methods are algebraic methods, logic-based approach, and graph-based. For $2D$ solvers, the graph-based approach has become dominant in CAD.

In practice, there exists hybrid techniques. For example, geometric constraint solving used in conjunction with equational methods, logic-based and graph based methods. See the works of Hoffmann [48], Joan Arinyo *et al.* [62], Gao *et al.* [37] and Fabre *et al.* [29].

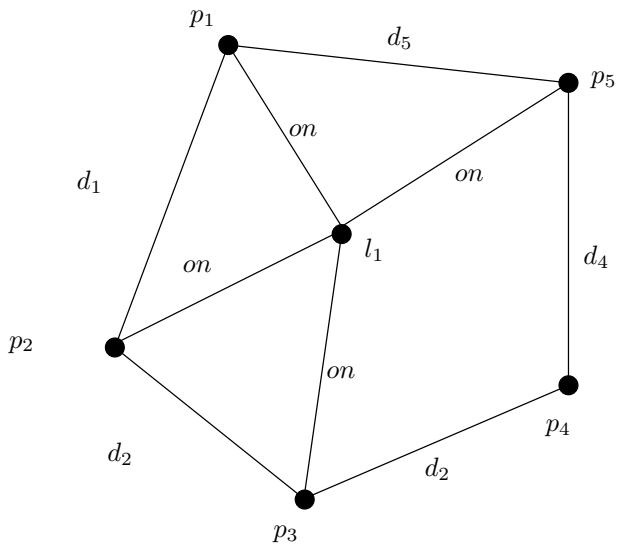


Figure 3.15: Geometric constraint graph associated to the crank-connecting rod problem in Figure 3.14(a).

3.3.1 Equational Methods

Equational methods accept as input a set of algebraic equations and can be further classified into numerical, symbolic and analysis of systems of equations.

The main advantage of equational solvers are their generality, dimension independence and the ability to deal with symbolic constraints naturally.

In principle, an equational solver can be fully competent. However, algebraic solvers may have low efficiency or may have difficulty constructing solutions reliably. When used to preprocess and study specific constraint problems, however, algebraic techniques can be extremely useful and very practical.

As a result of mapping the geometric domain problem into an equational one, the geometric sense of the solutions rendered is lost. Moreover, well constrained problems are mapped to under-constrained systems of equations because constraints fix the placement for each geometric element with respect to each other only modulo translation and rotation. Therefore a set of additional equations must be joined to cancel these remaining degrees of freedom.

Numerical Methods

Numerical methods are among the oldest methods of geometric constraint solving. Numerical methods provide powerful tools to solve iteratively large systems of equations. In general, a good approximation of the intended solution should be supplied to guarantee convergence. This means that if, as it is customary, the starting point is taken from the sketch defined by the user, then the sketch should be close to the intended solution. The numerical methods may offer little control over the solution in which the user is interested. To achieve robustness, numerical iterative methods must be carefully designed and implemented.

The method most widely used is the well-known *Newton-Raphson* iteration, [72]. It is used

in the solvers described in [42,87,88]. Newton-Raphson is a local method and converges much faster than relaxation. The method does not apply to consistently over-constrained systems of equations unless special provisions are made such as combining it with least-squares techniques.

Borning, [10], Hillary and Braid, [43], and Sutherland, [117], use a *relaxation method*. This method is an alternative to the propagation method. Basically, the method perturbs the values assigned to the variables and minimizes some measure of the global error. In general, convergence to a solution is slow.

Homotopy or *continuation*, [5] is a family of methods that guarantee convergence. Moreover, they are exhaustive and allow to determine all solutions of a constraint problem. However, their efficiency is worse than that of Newton-Raphson. Lamure and Michelucci, [82], Durand, [26], and Luo *et al.*, [90], apply this method to geometric constraint solving.

Other, less conventional, methods have also been proposed. For example, in [41], Hel-or *et al.*, introduced the *relaxed parametric design* method where the constraints are soft, that is, they do not have to be met exactly, the problem is modeled as a static stochastic process, and the resulting system of probabilistic equations is solved using the Kalman filter familiar from control theory. The Kalman filter was developed to efficiently compute linear estimators and when applied to nonlinear systems, it does not necessarily finds a solution even if one exists.

Symbolic Methods

Symbolic algebraic methods compute a Gröbner basis for the given system of equations. Algorithms to compute these bases include those by Buchberger, [18], and by Wu-Ritt, [20]. These methods, essentially, transform the system of polynomial equations into a triangular system whose solutions are those of the given system. In effect, triangularization reduces solving a simultaneous, nonlinear system to univariate root finding. Forward or a backward substitution must be used.

Buchanan *et al.*, [17], describe a solver built on top of the Buchberger's algorithm. In [74,75], Kondo improves that work by generating a polynomial that summarizes the changes undergone by the system of equations.

Analysis of Systems of Equations

Methods based on the analysis of systems of equations determine whether a system is under, over or well-constrained from the system structure. These methods can be extended to decompose systems of equations into a set of minimal graphs which can be solved independently, [94, 115]. They can be used as a preprocessing phase for any other method, reducing the number of variables and equations that must be solved simultaneously.

Serrano, [106], applies analysis of systems of equations to select from a set of candidate constraints a well constrained, solvable subsets of equations. Elber *et al.* [27] use multivariate rational functions.

3.3.2 Logic-Based Methods

Aldefeld, [4], Bruderlin, [13–16, 16] and Yamaguchi *et al.*, [134], use first order logic to derive geometric information applying a set of axioms from Hilbert's geometry. Essentially these methods yield the geometric loci at which the elements must be.

Sunde, [116], and Verroust, [126,127], consider two types of sets of constraints: set of points placed with respect to a local framework, and sets of straight line segments whose directions are fixed with respect to a local framework. The reasoning is basically performed by means of rewriting system on the sets of constraints. The problem is solved when all the geometric elements belong to a unique set. Joan-Arinyo and Soto-Riera, [62], extended these sets of constraints with a third type consisting of sets containing one point and one straight line such that the perpendicular point-line is fixed.

Theorem proving is a logic-based approach than can be seen as automatically proving a geometric theorem. However, automatic geometric theorem proving requires more general techniques and, therefore, methods which are much more complex than those required by geometric constraint solving.

Wu Wen Tsün pioneered the Wu-Ritt method, an algebraic-based geometric constraint solving method. In [130–132], he uses it to prove geometric theorems. The method automatically finds necessary conditions to obtain non-degenerated solutions.

In [20], Chou applies Wu's method to prove novel geometric theorems. Chou *et al.*, in [21] report on a work in automatic geometric theorem proving which allows to interpret, from a geometric point of view, the proof generated by computation.

3.3.3 Graph-Based Approach

In the graph-based approach, the constraint problem is translated into a graph (or hyper graph) whose vertices represent the geometric elements and whose edges the constraints upon them. The solver analyzes the graph and formulates a solution strategy whereby subproblems are isolated and their solutions suitably combined. A subsequent phase then solves all subproblems and combines them. The advantage of this type of solver is that the subproblems often are very small and fall into a few simple categories. The graph-based approach can be further subdivided into degree of freedom analysis, propagation and constructive approach.

Degree of Freedom Analysis

Degrees of Freedom Analysis assigns degrees of freedom to the geometric elements by labeling each edge of the graph with the number of degrees of freedom canceled by the associated constraint. Then the method solves the problem by analyzing the resulting labeled graph. In $2D$, a point would have 2 degrees of freedom, a circle 3. Each graph edge is labeled by the degrees of freedom canceled by the represented constraints. If the incident vertices are points in $2D$, for instance, an incidence constraint cancels 2 and a distance constraint cancels 1 degree of freedom. This graph is then analyzed for a solution strategy. To date, several approaches follow next.

Kramer, [77–79] developed a method to solve specific problems concerning the kinematics, specifically for the simulation of mechanisms. The method applies techniques borrowed from the process planning field to yield a symbolic solution. Since the set of rules used to generate the plan preserves geometric sense, the entire method also preserves it. Kramer, [77–79], proves that this method is correct by showing that the set of rules together with the labeled graph is a canonical rewriting system. The method runs in time $O(nm)$, where n is the number of elements geometry and m is the number of constraints in the problem. Since m is typically

$O(n)$, then the method has quadratic complexity. Bhansali *et al.*, [7], describe a method that generates automatically segments of the symbolic solution in Kramer's approach.

Salomons *et al.*, [102], represent objects and constraints using conceptual graphs and apply geometric and equational reasoning following the lines given by Kramer's method.

In [52,53], Hsu presents a method with two phases. First, a symbolic solution is generated. Then, the actual construction is carried out. The method applies geometric reasoning and, if this fails, numerical computation.

Latham *et al.*, [83], decompose the labeled graph in minimal connected components named balanced sets. If a balanced set corresponds to one of the predefined specific geometric constructions, then it can be solved. Otherwise the underlying equations are solved numerically. The method also deals with symbolic constraints and identifies over-constrained and under-constrained problems. Assigning priorities to the constraints allows them to solve over-constrained problems. A proof of correctness is also given.

Hoffman et al., [44–46], developed a flow-based method for decomposing graphs of geometric constraint problems. The method generically iterates to obtain a decomposition of the underlying algebraic system into small subsystems called *minimal dense* subgraphs. The method fully generalizes degree-of-freedom calculations, the approaches based on matching specific subgraphs patterns, as well as the prior flow-based approaches. However, the decomposition rendered does not necessarily have geometric sense since minimal dense subgraphs can be of arbitrary complexity far exceeding problems that yield to classical geometric construction.

Global Propagation Methods

Propagation methods represent the set of algebraic equations with a symmetric graph whose vertices are variables and equations and whose edges are labeled with the occurrences of the variables in the equations.

Propagation methods try to orient the edges in the graph in such a way that each equation vertex is a sink for all the edges incident on in except one. If the process succeeds, then there is a general incremental solution. That is, the system of equation can be transformed into a triangular system and solved using back substitution.

Among the techniques to orient a graph we find in the literature degrees of freedom propagation and propagation of known values, [31,103,125]. Propagation methods do not guarantee finding a solution whenever one exists. They fail when the orientation algorithm finds a loop. Propagation methods can be combined with numerical methods for equation solving to ameliorate circularity, [10, 80, 112, 116]. Veltkamp and Arbab, [125], apply other techniques to break loops created while orienting the graph.

Leler, [86], describes propagation methods in depth and proposes *augmented rewriting terms*, a tool which consists of a classical rewriting system along with an association of atomic-value and object-type. This tool has had success in solving certain systems of nonlinear equations.

In [11], Borning *et al.* describe a local propagation algorithm that can deal with inequalities.

Constructive Approach

The constructive approach generates the solution to a geometric constraint problem as a symbolic sequence of basic construction steps. Each step is a rule taken from a predefined set

of operations that position a subset of the geometric elements. For example, the operations may restrict to ruler-and-compass constructions. Clearly, this approach preserves the geometric sense of each operation involved in the solution. Note that the sequence of construction steps allows to compactly represent a possibly exponential number of solution instances. However, the constructive approach cannot solve problems with symbolic constraints or external variables.

This work focuses on the constructive approach. Thus we will discus it in depth in Section 3.4.

3.4 Graph-Constructive Solvers: Overview and Background

We will now turn to a description of constructive techniques to solve geometric constraint problems. We will first describe the most popular two techniques, decomposition and synthesis. After that, we will describe the tree decomposition technique used by Decomposition-Recombination solvers.

Constructive solvers have two major components: the analyzer and the constructor. The analyzer symbolically determines whether a geometric problem defined by constraints is solvable. If the answer is positive, the analyzer outputs a sequence of construction steps, known as *construction plan* that describes how to place each geometric element in such a way that all constraints are satisfied. After assigning specific values to the parameters, the constructor interprets the construction plan and builds an object instance, provided that no numerical incompatibilities arise, for example, computing the square root of a negative value.

The constructive approach generates the solution to a geometric constraint problem as a symbolic sequence of basic construction steps. Each step is a rule taken from a predefined set of operations that position a subset of the geometric elements. The specific construction plan generated by an analyzer depends on the underlying constructive technique and how it is implemented. For example, the ruler-and-compass constructive approach is a well-known technique where each constructive step in the plan corresponds to a basic operation solvable with a ruler, a compass and a protractor. In practice, this simple approach solves most useful geometric problems. Clearly, the ruler-and-compass approach preserves the geometric sense of each operation involved in the solution.

Note that the sequence of construction steps allows to compactly represent a possibly exponential number of solution instances. Generally, the user is only interested in one instance such that besides fulfilling the geometric constraints, exhibits some additional properties. This solution instance is the *intended solution* and selecting the intended solution amounts to selecting one solution instance among a number of different roots of a nonlinear equation or system of equations. The problem of selecting a given root is the Root Identification Problem.

However, the constructive approach cannot solve problems with symbolic constraints or external variables.

The constructive methods are classified depending on the technique used to analyze the problem. There are two cases, decomposition (top-down approach) and synthesis methods (bottom-up approach).

The top-down technique recursively splits the problem until it has isolated simpler, basic problems whose solutions are known. In this category, Todd, [124], defines the r-tree con-

cept and derives a geometric constraint solving algorithm. Owen, [99], described a top-down algorithm for computing a decomposition of an arbitrary constraint graph. The algorithm recursively splits the graph into split components. The algorithm terminates when the graph cannot be split further. At the end of the analysis the original graph has been decomposed into a set of basic subgraphs. The algorithm closely follows the triconnected components decomposition of [51]. In Joan-Arinyo *et al.*, [68], it is proved that the worst case running time complexity of this algorithm is $O(n^2)$.

Inspired by Owen's work, Fudos and Hoffmann, [35] reported on two graph-based constructive approaches to solve systems of geometric constraints. The top-down method is roughly equivalent to the method by Owen, [99]. The bottom-up method, named reduction analysis, begins by computing a set S of basic subgraphs. Then graphs in S are iteratively merged until a unique graph which contains all the geometric elements in the problem is obtained. Fudos and Hoffmann claim the algorithm to have a $O(n^2)$ runtime complexity in the worst case.

In the bottom-up approach, the solution is built by suitably combining recursively solutions to subproblems already computed, starting from the constraints in the given set, considering each constraint as a single element. Constraints may be represented implicitly as a collection of sets of geometric elements where the elements of each set are placed with respect to a local framework. Sets are merged; e.g., by application of rewriting rules until all the geometric elements are included in just one set. The advantage of this representation is that the sets of constraints capture the relationships between geometric elements compactly.

Fudos *et al.*, in the method described in [12,33–35], use one type of sets of constraints, called cluster, and one generic rule that merges three clusters which pairwise share two elements. Lee *et al.*, [84, 85], describe a constructive method that associates with each vertex in the graph a status which can be defined, half defined or not defined. Inference rules are used to modify the status of the vertices.

Efficient algorithms with a running time $O(n^2)$, where n is the number of geometric elements, are known for the methods listed above. However, the constructive approach is not complete, therefore assessing the competence of solvers in this category is an important issue. Verroust, [126, 127], partially characterizes the set of relevant problems solved by the solver described. Joan-Arinyo *et al.*, [65, 66],describe a formalization that unifies the methods reported by Fudos, [34, 35],and Owen, [99]. In [65, 68], Joan-Arinyo *et al.* show that the sets of problems solved by Fudos' and Owen's approaches are the same.

In [48] Hoffmann and Joan-Arinyo describe a technique that extends constructive methods with the capability of managing functional relationships between geometric and externals variables. Essentially, the technique combines two methods: one is a constructive constraint solving method, and the other is a systems of equations analysis method. Joan-Arinyo and Soto-Riera, [62], further improved the technique and formalized it as a rewriting system. Constructive methods work well in 2-space. Several attempts to extend them to 3-space have been reported. See, for example, Bruderlin , [13], Verroust, [126], and Hoffmann and Vermeer, [47].

Gao *et al.*,in [38], report on a method that from the constraint graph extracts a binary connectivity tree, called C-tree, according to a well defined set of rules. For each node in the C-tree, the left child represents a subproblem which can be solved. Then the subtree rooted in the right child is recursively visited and merged with the rigid body coming from the left child. The method is essentially equivalent to those described above.

Following the works of Joan-Arinyo *et al.*, [66, 128], the architecture for our constructive

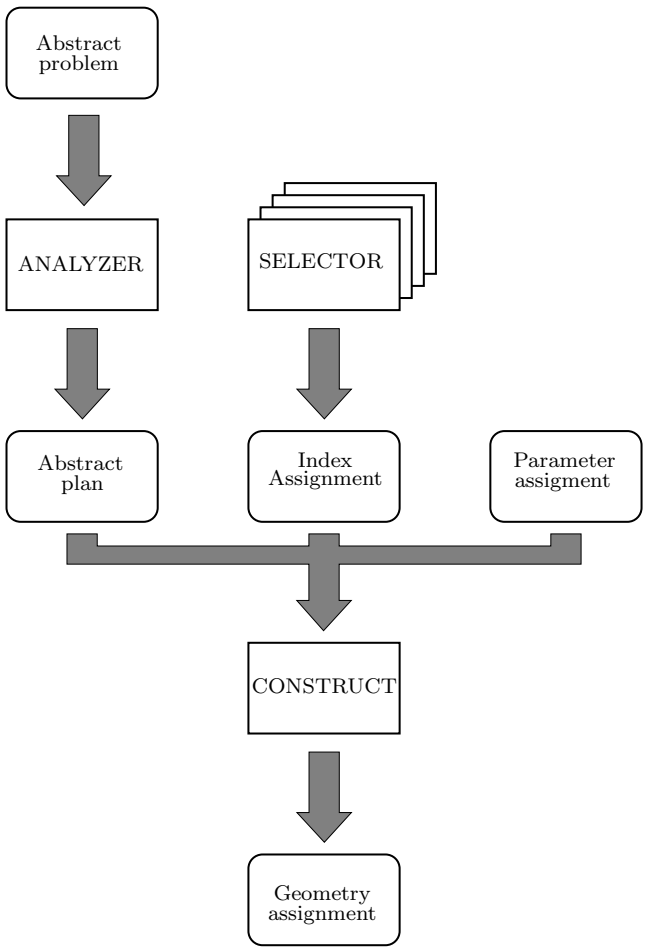


Figure 3.16: Architecture data flow diagram.

solver is illustrated in Figure 3.16. Square boxes are functional units and rounded boxes are data entities. The functional units include the *analyzer*, the *index selector* and the *constructor*. The data entities are the geometric constraint plan, the parameters assignment and the index assignment. Given a geometric constraint problem P, the analyzer is responsible for figuring out whether the solver is able to solve the problem, that is, whether it can find a description placement for the geometric objects. If the answer is positive, the analyzer outputs the construction plan, that will place the geometric elements in the position such that the constraints hold. An index selector can be seen generally as a functional unit characterized by its output which is an index assignment. An index assignment is always associated to a given abstract plan. Therefore the plan must also be considered an input to the index selector. For an in depth study of the index and the role it plays in a geometric constraint solving see [32].

Finally, once a set of actual values have been assigned to the constraint parameters, the constructor builds an instance of a placement for the geometric objects.

3.4.1 Decomposition-Recombination Solvers

In contemporary CAD systems constructive solvers the volume of items and geometric constraints between them is large, so it is used the strategy of divide and conquer resolution. This strategy consists of three basic steps, see Hoffmann *et al.*, [36], Ait-Aoudia, *et al.*, [1], which summarized as:

- Divide the problem P into subproblems $P_1, P_2, \ldots, p_n$.
- Solve each subproblem P_i in a recursive way with known algorithms.
- Construct a solution to P by merging solutions of the subproblems $P_1, P_2, \ldots, P_n$.

Constraint solvers that fit in this description are known as Decomposition-Recombination solvers (DR-solvers). Our constructive solver follow this pattern, and from now on we will refer to our solver as a DR-solver. DR-solvers approach has been particularly successful when the decomposition into sub-problems and the subsequent recombination solutions to these subproblems can be described by a plan generated a priori, that is, a plan generated as a preprocessing step without actually solving the subsystems. The output by the DR-planner remains unchanged as numeric values of parameters change. Such a plan is known as the DR-plan and the unit in the solver that generates it is the DR-planner. In this setting, the DR-plan is then used to drive the actual solving process, that is, computing specific coordinates that properly place geometric objects with respect to each other. Then, the DR-plan specifies a plan for decomposing a constraint problem into small subproblems and recombining solutions of these subproblems later.

Our interest focuses on DR-solvers where the geometric constraint problem is abstracted as a graph and thus, solving the problem generically corresponds to solving the corresponding graph. Therefore the DR-planner is based on graph analysis whose operation will be purely combinatorial. We will consider well-constrained problems, loosely speaking, problems where resulting geometric objects are well defined.

In a pioneering work, Hoffmann et al., [45], classify graph-based DR-planners into two main categories. Planners in one category, called Constraint Shape Recognition, work by recognizing specific solvable subgraphs of known shape, most commonly, triangular patterns. DR-planners have been widely used and related literature is profuse. Examples are [12, 16, 25, 35, 38, 55, 60, 91, 98–100, 105, 127]. DR-planners in the second category are called Generalized Maximum Matching planners. Planners in this category work in two steps. First they isolate certain solvable subgraphs by transforming the constraint graph into a bipartite graph where a maximum generalized matching is found. Then the DR-plan is figured out by connectivity analysis. To this category belong DR-planners reported in [1, 2, 57, 79, 83].

In [44], Hoffmann et al. reported on two new DR-planners that exhibit features of both categories. The Condensed Algorithm applies repeatedly a flow-based algorithm to find minimal dense solvable subgraphs, [46], that are subsequently extended by adding more geometric objects one at a time. The Frontier Algorithm and the Modified Frontier Algorithm are essentially improvements of the Condensed Algorithm. These algorithms have been further developed for problems considering purely distance constraint systems by Lomonosov, [89] and by Zhou, [137]. Surveys can be found in [6, 58]. A discussion on geometric constraint solving progress and research directions can be found in [108].

More recent research in this area includes the following other aspects such as, obtaining an optimal and canonical decomposition. Lomonosov, [89] show NP completeness of the optimal DR-planning problem even for 2D. Then, define a notion of a canonical decomposition into a complete collection of proper vertex-maximal rigid subgraphs, and completely solve the canonical decomposition problem for 2D distance constraint systems. First for well-constrained systems (where a canonical decomposition is unique), then for over-constrained systems, where it is not unique. Zhou, [137] characterizes rigidity for 2D angle constraint systems. The works of Sitharam, allow to obtain an optimal parametrization from recombination, [111]. Finally, we cite works that use DR-plan for other applications, such as overconstrained detection, Hoffmann *et al.*, [50], solution navigation, Sitharam *et al.*, [109], feature-based, Sitharam *et al.*, [110] or modeling virus assembly, Sitharam *et al.*, [8].

Our DR-planner follows the approach to divide the geometric constraint problem into subproblems based on the tree decomposition method. The tree decomposition of a constraint graph was introduced by Joan-Arinyo *et al.*, in [68, 70]. This concept is figured out in detail in next section.

3.5 Tree Decomposition of a Geometric Constraint Graph

Following the work of Joan-Arinyo *et al.* in [68], this section defines the concepts of *decomposition of a set* and *tree-decomposition*. These concepts are the base of the construction technique for solving systems of geometric constraints, since the calculation of a tree decomposition directly defines a constructive plan. Then we present and illustrate the main concepts associated with the constraint graph decomposition, which are the basis of the constructive method that we develop.

Recall we consider constraint problems in the Euclidean plane defined by giving a set of geometric element like points, lines, line segments, circles and circular arcs, along with a set of relationships, called *constraints*, like point-point distance, angle between straight lines, incidence and tangency.

The concept of *set decomposition* refers to a way of partitioning a given abstract set. Next, we define the concept of *tree decomposition* of a graph.

Let S be a set, with at least, three different members, say a, b, c. We say that three subsets of S, say S_1, S_2 and S_3 is a *set decomposition* of S if the following properties are fulfilled.

1. $S_1 \cup S_2 \cup S_3 = S$
2. $S_1 \cap S_2 = \{a\}$
3. $S_2 \cap S_3 = \{b\}$
4. $S_1 \cap S_3 = \{c\}$.

Vertices $\{a, b, c\}$ are called the *hinges* of the set decomposition.

Example 3.12. Figure 3.17 shows a set and one possible set decomposition.

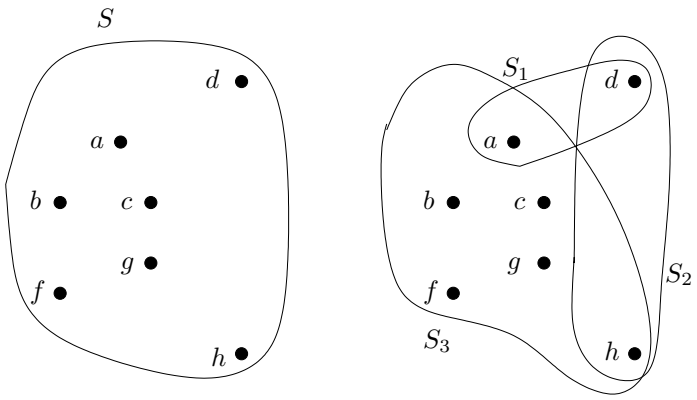


Figure 3.17: (a) Set S with three or more members. (b) Set decomposition of S.

The *set decomposition* of a graph is defined as follows. If $G = (V, E)$ is a constraint graph, the subsets $V_1(G), V_2(G)$ and $V_3(G)$ define a set decomposition of G if they are a set decomposition of $V(G)$ and for every edge $e = (v_1, v_2)$ with $e \in E(G)$, $v_1, v_2 \in V(G_i)$, for some i, $1 \leq i \leq 3$. Roughly speaking, a set decomposition of a graph G, is a set decomposition of the set of vertices $V(G)$ such that the two ends of each edge belong to a given subset of vertices.

Next, we illustrate the concept of set decomposition.

Example 3.13. Figure 3.18 shows a graph $G = (V, E)$ and a non-decomposition set of V due to the fact that vertices incident to edge (e, b) do not belong to any set in the partition.

Example 3.14. Consider the geometric constraint problem in Figure 3.19. The geometric elements are: $O = \{A, B, C, f, d\}$, the parameters of the constraints are $P = \{d_1, h_1, \alpha\}$, and the set of constraints is:

r1: $dpl(A, e) = h_1$,
r2: $on(B, e)$,
r3: $dpp(A, B) = d_1$,
r4: $ang(d, e) = \alpha$,
r5: $on(A, d)$,
r6: $on(C, e)$,
r7: $on(C, d)$,
r8: $on(B, f)$,
r9: $on(A, f)$

Figure 3.20(a) shows the graph representation of the problem in Figure 3.19, and Figure 3.20(b) shows the corresponding decomposition.

A naive approach to compute hinges is to perform an exhaustive search of subsets of three vertices of the constraint graph that allow a set decomposition of the input graph. This amounts to considering different combinations of three different vertices of V until finding one tern that allows to split the graph G in three disjoint clusters G_1, G_2 and G_3. The number of

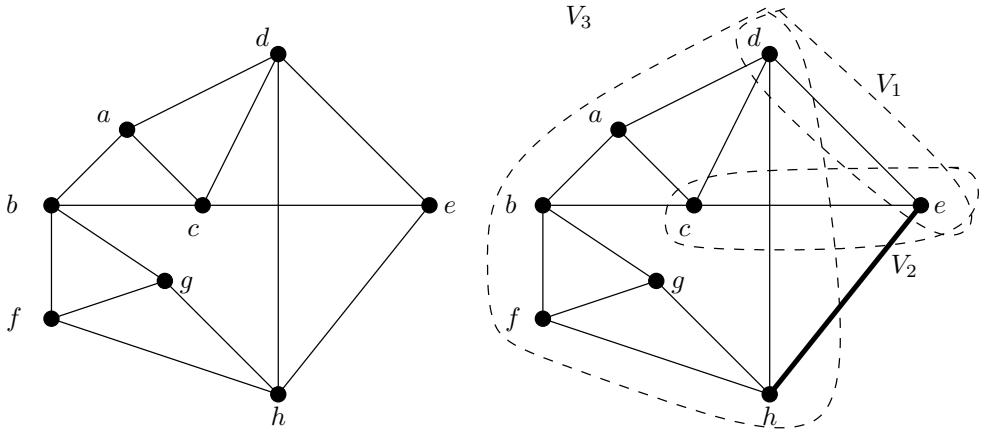


Figure 3.18: (a) G. (b) A non-decomposition set of G (broken edge).

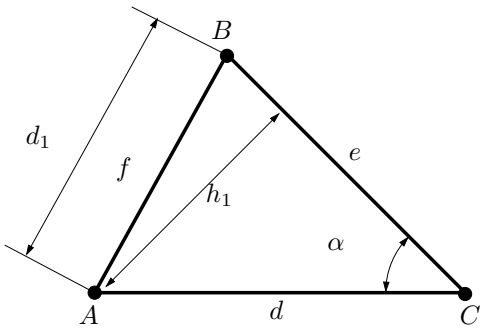


Figure 3.19: A Geometric constraint problem.

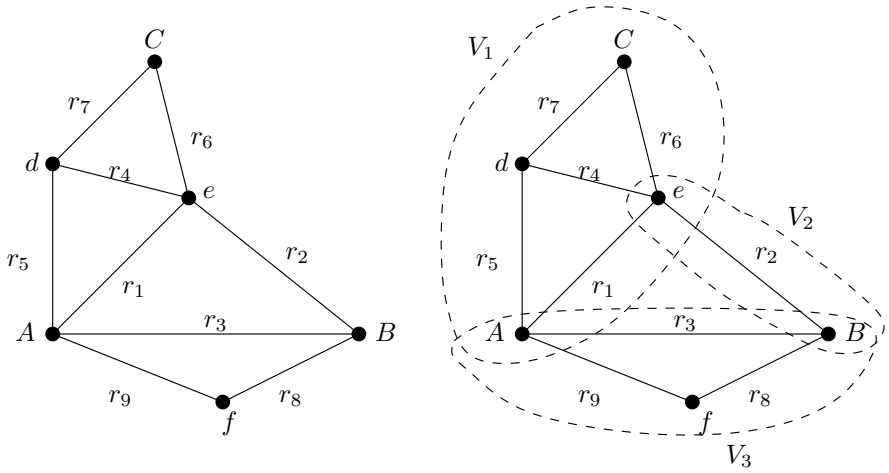


Figure 3.20: (a) Graph G. (b) Set decomposition of G.

Algorithm 1 Computing a set decomposition by means of exhaustive search.

INPUT
A constraint graph $G = (V, E)$ with $|V| \geq 3$
OUTPUT
A set decomposition of G, if exists
forevery $\{a, b, c\} \subseteq V$ **do**
Compute $V' = V - \{a, b, c\}$
forevery $\{G'_1, G'_2, G'_3\}$ partition of V' **do**
Compute $G_1 = G'_1 \cup \{a, c\}$
Compute $G_2 = G'_2 \cup \{a, b\}$
Compute $G_3 = G'_3 \cup \{b, c\}$
if $\forall e \in E : V(e) \subseteq G_1$ or $V(e) \subseteq G_2$ or $V(e) \subseteq G_3$ **then**
return $\{G_1, G_2, G_3\}$
done
done
return $\emptyset$

combinations for a graph G is $\binom{|V|}{3}$. The algorithm for calculating the set decomposition of a constraint graph by exhaustive searching the hinges is shown in Algorithm 1.

Finally we define the concept of *tree decomposition* of a graph. Let $G = (V, E)$ be a graph. We say that a ternary tree T is a *tree decomposition* of G if

1. $V(G)$ is the root of T,
2. Each node $V' \subset V(G)$ 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 $V'(G)$, and
3. Each leaf node contains exactly two vertices $\{a, b\}$ of V such that edge (a, b) is in $E(G)$.

A graph for which there is a tree decomposition is called *tree decomposable*. In general, a tree decomposition of a graph is not unique. If a graph is tree decomposable, then any subgraph obtained by removing some of its edges is also tree decomposable.

Theorem 3.2. *Let $G = (V, E)$ be a tree decomposable graph. For all $E' \subseteq E$, the subgraph of G with the set of edges E', $G' = (V, E')$, is tree decomposable.*

Proof. See Joan-Arinyo *et al.*, [49]. □

Example 3.15. Figure 3.21 represents the tree decomposition of graph in Figure 3.20. Figure 3.22 shows the representation over the constraints graph of the tree decomposition of Figure 3.21.

Essentially, the decomposition tree T aim is to calculate a *construction plan*. A construction plan specifies precisely how to solve the geometric constraint problem represented by the graph G.

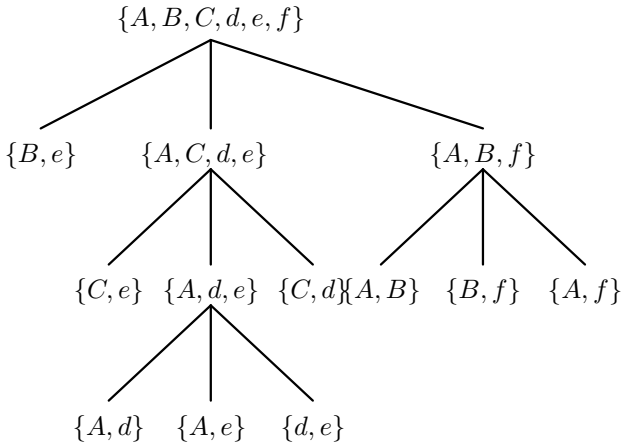


Figure 3.21: Tree decomposition of graph in Figure 3.20a.

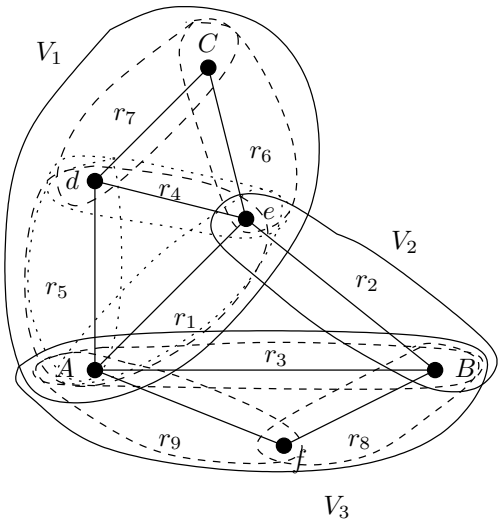


Figure 3.22: Recursive decomposition of graph G in Figure 3.20a.

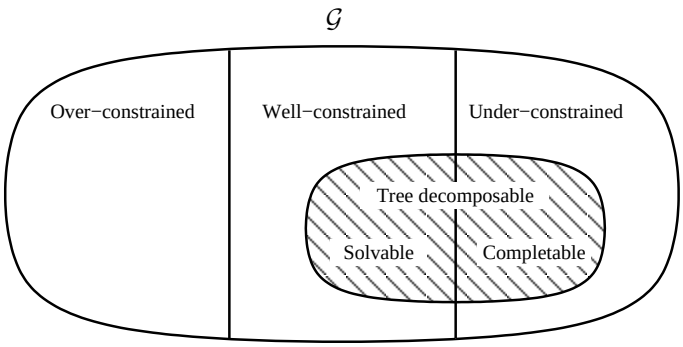


Figure 3.23: Classification of geometric constraint graphs, [67].

3.5.1 Tree Decomposable Graphs

In this section, given a geometric constraint problem represented by a geometric constraint graph, we describe the relationship between tree decomposable graphs and solving under- well- constrained problems.

Assume that $G = (V, E)$ denotes the set of geometric constraint graphs. The definitions of over-, well and under-constrained graphs induce a partition on G. Following the works of Joan-Arinyo *et al.*, [67], see Figure 3.23, we note that 1) the tree decomposable graphs are a subset of the graphs which are not over constrained (See Theorem 3.3), 2) the class of tree decomposable graphs overlaps the set of well constrained graphs and the set of under-constrained graphs, and 3) the solvable graphs (or full tree decomposable graphs) are those well constrained graphs which are tree decomposable (See Theorem 3.4).

Theorem 3.3. *Let $G = (V, E)$ be a geometric constraint graph. If G is tree decomposable, then G is not over-constrained.*

Proof. See [67]. □

Theorem 3.4. *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. See [67]. □

3.6 Construction Plan

From the tree decomposition of a graph of geometric constraints we retrieve a construction plan. To calculate it we perform a postorder tour of the nodes of the tree. Each step positions a new element and it finishes having positioned all vertices of the graph. We illustrate the procedure with a couple of examples.

Let $G = (V, E)$ is the graph of an abstract geometric constraint problem, a tree decomposition of G can be seen as a construction plan that solves the problem at hand. Each node in the tree stands for a rigid object, called *cluster*, built on the geometric objects included in the node. Leaf nodes represent elemental placement problems corresponding to two geometric

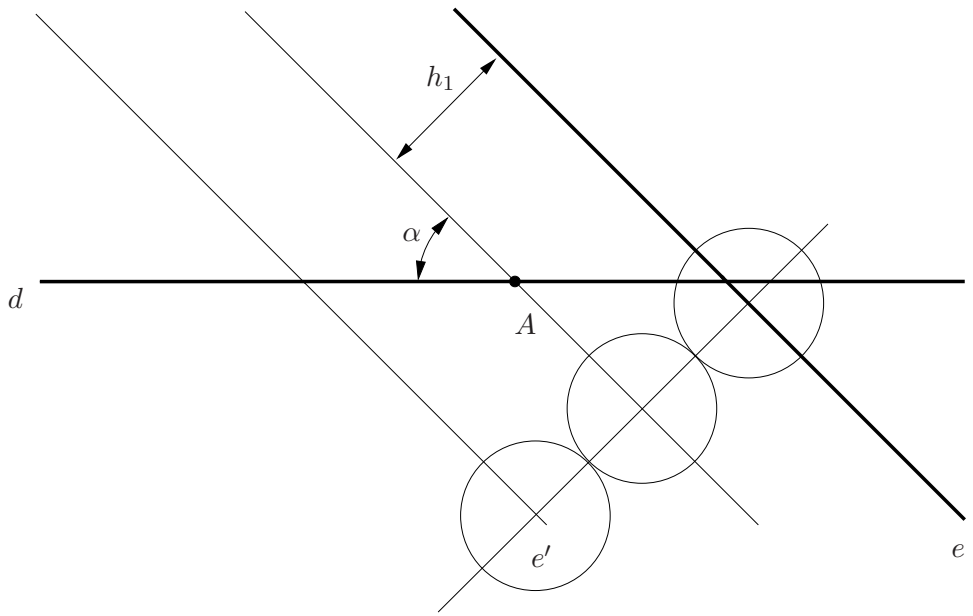


Figure 3.24: Construction of a part of the plan of the tree decomposition in Figure 3.21.

elements and the constraint defined on them. Examples are: two points at a given distance, a point and a straight segment at a given distance, two straight segments at a given angle and so on.

Each internal node in the tree decomposition represents the merging of three solved clusters into a larger rigid cluster by application of a specific solving rule. The root node includes all the geometric elements in the problem and represents a solution instance. Next we illustrate the procedure.

Example 3.16. Consider the Figure 3.21 that represents the tree decomposition of graph in Figure 3.20a. A postorder route could begin by the leaves, $\{A, d\}$, $\{A, e\}$ and $\{d, e\}$. This would generate a fragment of plan construction as

$$\begin{aligned} d &= line() \\ A &= pointover(d) \\ aux_1 &= linePointAngle(A, \alpha, d) \\ e &= parallel(aux_1, h_1) \end{aligned}$$

that satisfy the constraints $r5$, $r1$ i $r4$.

In this plan, the operations are geometric operations with ruler and compass. For example, operation $pointover(d)$ locates the point A on the line d, and the operation $linePointAngle(A, \alpha, d)$ draws a line through the point A and form an angle α with line d. Figure 3.24 shows the construction of this part of the plan.

Following the path, we reach the leaves $\{C, E\}$ and $\{C, d\}$, which merged with $\{A, d\}$. As a

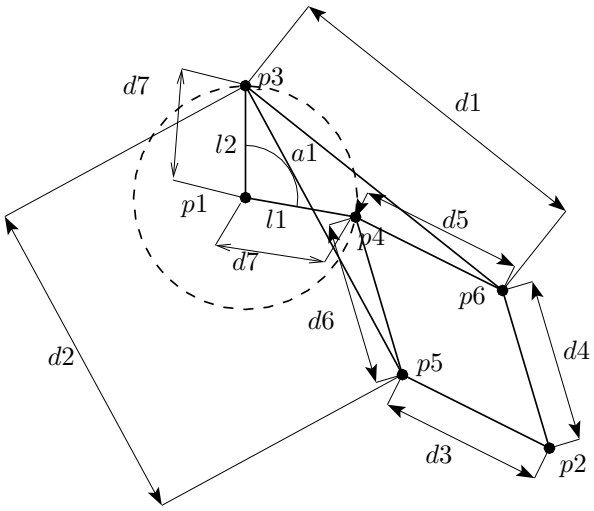


Figure 3.25: Peaucellier's linkage.

result, we obtain the positioning of point C. The portion of the plan that do the calculation is:

$$C = intersection(d, e)$$

Proceeding in analogy with the rest of the path ends obtaining the following plan:

$$\begin{aligned} d &= line() \\ A &= pointover(d) \\ aux_1 &= linePointAngle(A, \alpha, d) \\ e &= parallel(aux_1, h_1) \\ C &= intersection(d, e) \\ f &= line() \\ A &= pointover(f) \\ B &= pointLineDistance(f, d_1) \\ aux_2 &= circle(A, d_1) \\ B &= intersection(e, aux_2) \\ f &= line(B, A) \end{aligned}$$

Example 3.17. Following, [68,135], Figure 3.25 shows a mechanism known as the Peaucellier's linkage, that transforms the circular motion of point p_3 along circle c_1, into the translation of point p_2 along the straight line l_3. As a geometric constraint problem, Peaucellier's linkage includes six point, p_i, $1 \leq i \leq 6$, and two straight segments l_1, l_2. The set of geometric constraints are those listed in Figure 3.26 which includes point-point distance constraint, $dpp()$, coincidence, $on()$, and angle between two straight segments, $angle()$.

1. $dpp(p3, p6, d1)$
2. $dpp(p3, p5, d2)$
3. $dpp(p2, p5, d3)$
4. $dpp(p2, p6, d4)$
5. $dpp(p4, p6, d5)$
6. $dpp(p4, p5, d6)$
7. $dpp(p4, p1, d7)$
8. $dpp(p3, p1, d7)$
9. $on(p1, l1)$
10. $on(p4, l1)$
11. $on(p1, l2)$
12. $on(p3, l2)$
13. $angle(l1, l2, a1)$

Figure 3.26: Geometric constraints for Peaucellier's linkage in Figure 3.25.

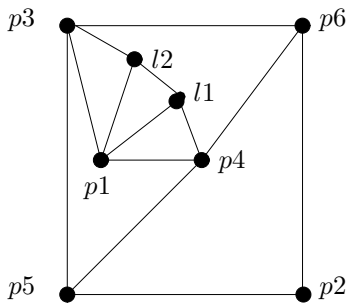


Figure 3.27: Geometric Constraint graph for Peaucellier's linkage in Figure 3.25.

1. $p1 = pointXY(0, 0)$
2. $p3 = pointXY(0, d7)$
3. $l2 = line2P(p1, p3)$
4. $l1 = lineAP(l2, p1, -a1)$
5. $x1 = circleCR(p1, d7)$
6. $p4 = ilc(l1, x1, s1)$
7. $x2 = circleCR(p3, d2)$
8. $x3 = circleCR(p4, d6)$
9. $p5 = icc(x2, x3, s2)$
10. $x4 = circleCR(p3, d1)$
11. $x5 = circleCR(p4, d5)$
12. $p6 = icc(x4, x5, s3)$
13. $x6 = circleCR(p5, d3)$
14. $x7 = circleCR(p6, d4)$
15. $p2 = icc(x6, x7, s4)$

Figure 3.28: Construction plan for Peaucellier's linkage in Figure 3.25.

Figure 3.27 shows the geometric constraint graph for the problem in Figure 3.25.

Figure 3.28 shows the construction plan generated by the constructive ruler-and-compass solver reported in for the Peaucellier's linkage in Figure 3.25. Geometric operations included in the plan are standard ruler-and-compass operations intersection between lines and points, construction of straight lines, construction of circles and so on. Function names in the plan are self explanatory. For example, function *line2P* defines a straight line through two points, *circleCR* defines a circle by its center and radius, and functions *ilc()*, *icc()* denote line-circle intersection and circle-circle intersection, respectively. Parameter s_i identifies one out of the two possible intersections. The index in Figure 3.28 is $I = \{s_1, s_2, s_3, s_4\}$.

3.7 Chapter Summary

In this Chapter we have defined the geometric constraint problem and we have classified it as a well constrained, over-constrained or under-constrained problem. Then, we reviewed the main approaches to represent and solve geometric constraint problems. The well constrained concept has been illustrated with two example: the crank shaft- connecting rod and the elevator door mechanism.

We focused on the constructive graph-based approach. In general algorithms reported in the literature solve the constraint problem by decomposing the constraints graph after identifying subgraphs according to a set of predefined rules or by applying flow propagation techniques. We described in detail the tree decomposition technique and illustrated the process of obtaining the constructive plan derived from the tree decomposition.

CHAPTER 4

A new DR-planner

> Good code is short, simple, and symmetrical —the challenge is figuring out how to get there.
>
> *(Sean Parent)*

Geometric constraint-based is the base technology in parametric solid modeling and its applications to Computer-Aided Design Software. Geometric constraint solving is the field devoted to solve geometric constraint problems. Among the techniques available in the literature one of the most most widely used is the constructive approach of decomposition-recombination methods. As we have seen in Chapter 3, in this work we focus on the constructive graph-based approach. In this chapter we describe a new DR-planner algorithm which computes a construction plan that solves the geometric constraint solving problem as a tree-decomposition of the geometric constraint graph received as input. The new algorithm first computes a set of fundamental circuits in the given constraint graph, then splits the graph into a set of subgraphs each sharing exactly three vertices with one fundamental circuit. The procedure is recursively applied to each subgraph until reaching a trivial problem which is solved by an specific dedicated solver.

Most of the work of this chapter has been previously published in references [69, 70].

4.1 Overview

Our approach to solve the geometric constraint solving problem proceeds as shown in Algorithm 2. This approach considers three basic different cases depending on the graph connectivity. For 0 and 1 connected graphs, there is a smooth approach while for biconnected graphs, the approach is far more difficult, [63]. In what follows we describe our solution for each case.

4.2 Decomposing 0-connected Graphs

We start by introducing some notation. Then, we exemplify the procedure and briefly show the decomposition algorithm for 0-connected constraint graphs.

Algorithm 2 The general new constructive tree-decomposition based *DR-planner*.

procedure decomp
INPUT
A well constrained graph $G = (V, E)$ with $|V| > 3$
OUTPUT
A tree decomposition T of G
if $0 - connected(G)$ **then**
$G_1, G_2, G_3 = 0 - con - decomp(G)$
else $1 - connected(G)$ **then**
$G_1, G_2, G_3 = 1 - con - decomp(G)$
else $biconnected(G)$ **then**
$G_1, G_2, G_3 = biconn - decomp(G)$
endif
T = tree(decomp(G_1), decomp(G_2), decomp(G_3))

Let $G = (V, E)$ be a 0-connected constraint graph with $|V| > 3$. Let $\mathcal{K} = \{K_1, \ldots, K_n\}$ be the set of connected components of G. Then $|K| \geq 2$ because G is 0-connected. We compute the hinges in G following this strategy (see Figure 4.1):

1. Choose a non-trivial subset of $\mathcal{K} \supset \{K_1, \ldots, K_m\}$ in such a way that, $G_1 = \sum_{i=1}^{m} K_i$ has at least two vertices $v_1, v_2 \in V(G_1)$ and $v_1 \neq v_2$. Note that this subset always exists.

2. Let $G_2 = \mathcal{K} - \{K_1, \ldots, K_m\} \cup v_1$. Let $v_3 \in V(G_2)$ be an arbitrary vertex of G_2.

3. Let G_3 be the graph with $v_2, v_3 \in V(G_3)$. Then v_1, v_2, v_3 are hinges of G.

Example 4.1. Figure 4.3 shows a 0-connected graph with hinges $\{c, e, a\}$.

Example 4.2. Figure 4.2 shows a 0-connected graph with hinges $\{e, d, a\}$.

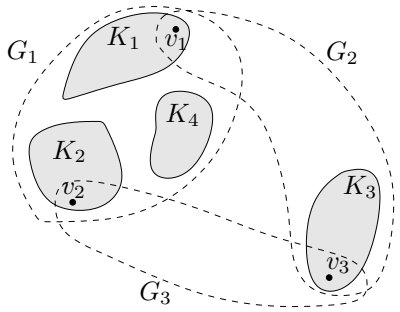


Figure 4.1: Decomposition of 0-connected graphs.

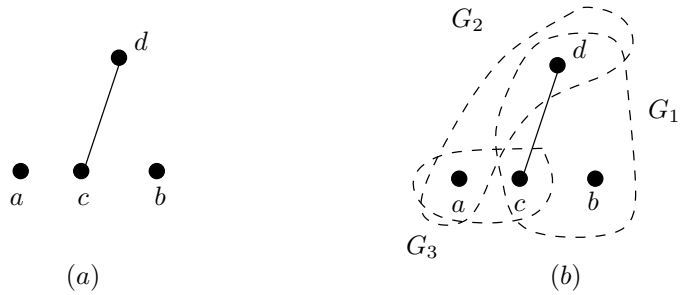


Figure 4.2: (a) A 0-connected graph and (b) Hinges $\{a, c, d\}$ of the set decomposition G_1, G_2 and G_3.

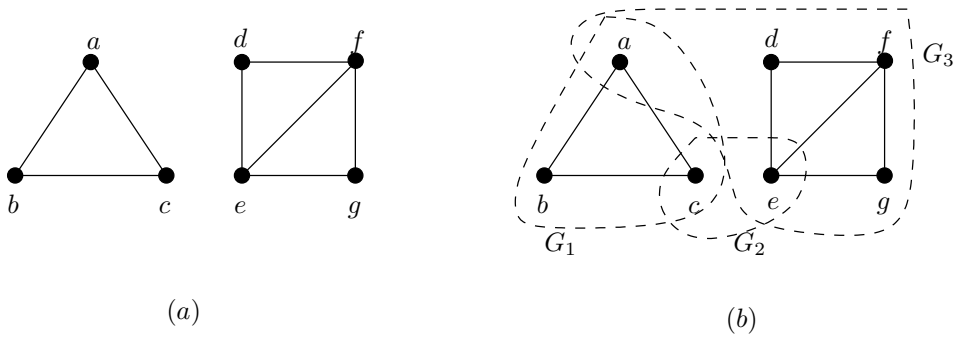


Figure 4.3: (a) A 0-connected graph G and (b) Hinges $\{c, e, a\}$ of the set decomposition G_1, G_2 and G_3.

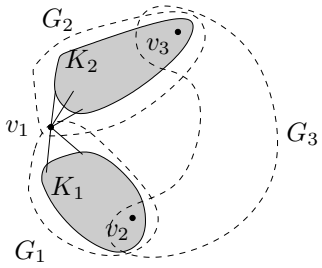


Figure 4.4: Decomposition of 1-connected graphs.

4.3 Decomposing 1-connected Graphs

Not only 0-connected graphs have a trivial decomposition algorithm but also 1-connected constraint graphs. Note that for the case of 1-connected graphs, there is just one connected component with one or more articulation vertices.

Let $G = (V, E)$ be a 1-connected constraint graph with $|V| > 3$. To compute the hinges in G we proceed as follows (see Figure 4.4),

1. Let v_1 be an articulation vertex of $V(G)$. Note that v_1 must exist because G is 1-connected.
2. Let $K = \{K_1, \ldots, K_m\}$ be a set of connected components of the graph $V(G_1) = V(G) - \{v_1\}$. $|K| \geq 2$ because v_1 is an articulation vertex. Arbitrarily select two connected components of K, say $K_1 = (V_1, E_1)$ and $K_2 = (V_2, E_2)$. Let $v_2 \in V_1$ and $v_3 \in V_2$.
3. Then $\{v_1, v_2, v_3\}$ are hinges of G.

Example 4.3. Figure 4.5 shows a 1-connected graph with hinges $\{c, e, g\}$.

4.4 Decomposing Biconnected Graphs

In this section we describe an algorithm to compute a set decomposition of a biconnected constraint graph $G = (V, E)$. Furthermore, we focus on decomposing graphs which capture well-constrained geometric problems. With the aim of improving the algorithm performance, we consider first the case where the given graph includes vertices with degree two. Then we shall consider graphs with degree three or higher.

4.4.1 Decomposing Graphs with Degree 2 Vertices

Todd, [124], reported on a method for subdividing a graph $G = (V, E)$ with degree 2 vertices. In this approach, graphs are subdivided by isolating vertices of degree two from their neighbors. This subdivision method is rather limited but can be satisfactorily combined with techniques for decomposition of a graph. Graphs simplification by means of deleting vertices with degree 2 is applied as follows:

1. Let $G = (V, E)$ a graph and $v \in V$ a vertex with degree 2 and consider $(v, v_1) \in E$ and $(v, v_2) \in E$.

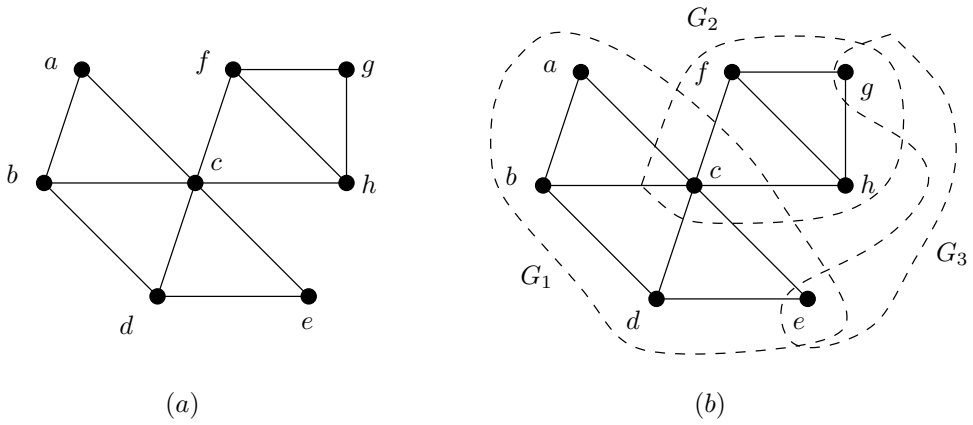


Figure 4.5: (a) A 1-connected graph and (b) Hinges $\{c, e, g\}$ of the set decomposition G_1, G_2 and G_3.

2. Then G can be decomposed using v,v_1 and v_2 as hinges and considering the clusters $G_1 = \{v, v_1\}$,$G_2 = \{v, v_2\}$, $G_3 = V - \{v\}$.

Todd decomposition method is shown in Figure 4.6. This decomposition technique can be efficiently implemented if the graph is stored as an adjacency list where the set of vertices with degree 2 can easily identified.

Example 4.4. Figure 4.7 shows the method applied to the graph G corresponding to the elevator problem in Figure 3.13 in Chapter 2. Note that the decomposition is reached by recursively removing degree 2 vertices.

In our approach, there is a preprocessing step, which trivially solves constructions corresponding to geometric elements whose nodes in the constraint graph have degree two, that is, geometric elements on which just two constraints have been defined. After sequentially removing degree two vertices, the connectivity of the graph to be further considered is two or higher.

4.4.2 Decomposing Graphs with Higher Degree Vertices

Now we focus now on decomposing biconnected constraint graphs whose vertices have degree three or higher. The main idea underlying the algorithm is that if there is a planar embedding

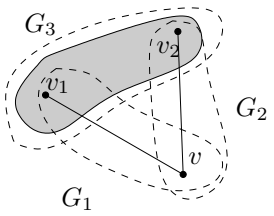


Figure 4.6: Decomposition based in a 2-degree vertex.

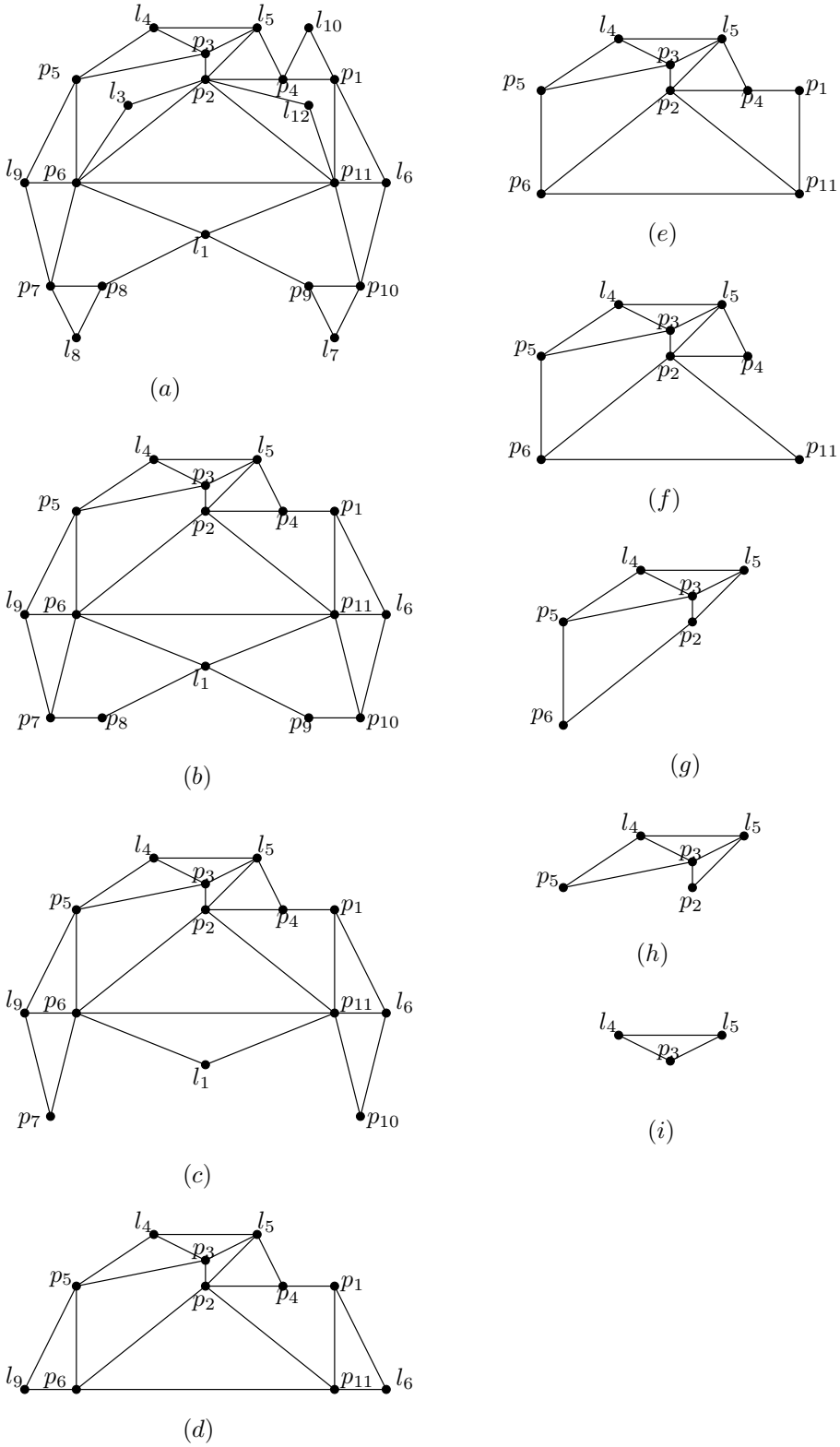


Figure 4.7: Decomposition by deleting degree 2 vertices of the Graph G corresponding to the elevator problem in Figure 3.13.

of either the graph G or a graph resulting from a specific transformation of G, where two faces share three vertices say $\{a, b, c\}$, then these vertices are hinges for G that define a set decomposition of $V(G)$. Moreover, a recursive application of this algorithm yields a decomposition of a tree decomposable graph. We start by recalling some graph transformations from general theory of graphs that we will use later on in the algorithm.

4.4.3 The Algorithm

Now follows our algorithm to compute a set decomposition of a biconnected constraint graph $G = (V, E)$. We need to solve in an efficient way the following problem,

> Given a biconnected constraint graph $G = (V, E)$, with a fundamental circuit C, compute the hinges contained in C, if any.

The new algorithm is based on the decomposition of a graph $G = (V, E)$, according to the bridges induced in G by a circuit C, and is inspired in the algorithm for finding the triconnected components of a graph reported by Miller and Ramachandran, [92, 93]. If $G = (V, E)$ is the geometric constraint graph, the algorithm starts by computing a spanning tree for G using a depth-first search and, from it, computing the set of induced fundamental circuits. Then, the algorithm seeks for a fundamental circuit C with a set of hinges that define a set decomposition of $V(G)$. Then, given a biconnected geometric constraint graph $G = (V, E)$ derived from a geometric constraint problem, the approach to compute a set decomposition of G is as follows:

1. Compute a spanning tree for the graph G by applying a depth-first search. Then the associated fundamental circuits $\{C_1, \ldots, C_n\}$ are identified. The goal is to find one fundamental circuit in which the hinges are contained in, as we will show later by means of Theorem 5.1.

2. Let $C \in \{C_1, \ldots, C_n\}$ be an arbitrary fundamental circuit. We compute the set of bridges $B = \{B_1, \ldots, B_k\}$ of G with respect to C.

3. The bridges of G are transformed into stars. Now the *collapsed graph* G' can be computed.

4. In the graph G' the bridges B' are merged according to the procedure of merging stars bridges. The result is the *merged graph* G''.

5. Now, all the bridges in $B_i \in B''$ are non interlacing star graphs, and star graphs are planar. As a result, Theorem 2.6 applies. Then we perform a particular planar embedding of G''. Therefore, given G'', we compute a *planar graph* G'''. G''' divides the plane in a set of faces $F = \{f_1, \ldots, f_m\}$. Each face $f \in F$ has a boundary formed by vertices and edges.

6. We will show on next Chapter 5 that if there are two faces $f_i, f_j \in F$ such that $f_i \cap f_j = \{v_a, v_b, v_c\} \subset V(G)$, then $\{v_a, v_b, v_c\}$ are hinges that define a set partition of $V(G)$ and therefore decompose G into three clusters. Then, we search in the planar graph for two faces $f_i, f_j \in F$ sharing almost three vertices. In our algorithm we fix the bounded face f_b, defined by the circuit under consideration, as one of the two faces to be checked.

Algorithm 3 The basic tree-decomposition algorithm for the DR-planner.

procedure biconn-decomp
INPUT
$G = (V, E)$: Biconnected constraint graph with $|V| > 3$
OUTPUT
$\{G_1, G_2, G_3\}$: a set decomposition of G, if one exists
Compute an spanning tree T of G
Compute the set of fundamental circuits C of G according to T
for C_i in C **do**
Compute the set of bridges B induced in G by C_i
Compute the collapsed graph G'
Compute the merged graph G''
Compute F, a planar embedding of G''
$hinges = findHinges(F)$
Compute the set decomposition G_1, G_2, G_3 induced by $hinges$
return G_1, G_2, G_3
endfor
return $\emptyset$

If the algorithm search of hinges fails for one fundamental circuit, the entire procedure is repeated until all the fundamental circuits are analyzed. Once one set of hinges if found, we recursively apply the algorithm to each of the clusters G_1, G_2, G_3 induced by hinges $\{v_1, v_2, v_3\}$ to obtain the complete tree decomposition. In case the algorithm fails in finding a fundamental circuit with a set of hinges, the input graph is not tree decomposable.

Algorithm 3 shows the basic steps in our algorithm to compute one decomposition step by splitting a graph into three connected components. Notice that the algorithm returns an empty set if it does not find any triple of vertices such that splits the graph. Clearly this means that the graph is not tree decomposable.

Example 4.5. The steps of the new algorithm are detailed next together with a set of examples. To illustrate the entire procedure, we consider the graph G in Figure 4.8 as the input of the algorithm.

4.4.4 Computing the Set of Fundamental Circuits

In Chapter 2 we provided definitions and methodology for finding fundamental circuits in a graph G. In this section we illustrate the procedure. We start computing the set of fundamental circuits of G, as we explained in section 2.5 by applying a depth-first search technique. To identify a set of fundamental circuits, first a spanning tree T to the constraint graph G is computed using a depth first spanning tree algorithm, [123].

Example 4.6. Figure 4.8 is the input graph $G = (V, E)$. If the starting arbitrary root is vertex a, Figure 4.9(a) shows the resulting depth first spanning tree T of G where edges in T are in bold lines and chords are drawn as dotted lines. Notice that every chord in the

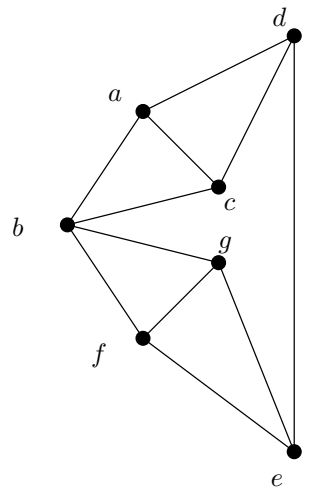


Figure 4.8: Case study graph.

spanning tree induces a fundamental circuit. Figure 4.9(b) illustrates the procedure to obtain the fundamental circuits of G according to a spanning tree T of G.

4.4.5 Computing the Set of Bridges

Let G be a biconnected graph and let C be a fundamental circuit of G. For our purposes, given a constraint graph $G = (V, E)$ and a circuit $C \subseteq G$, identifying a set of *bridges* is paramount, following Chapter 2.7.

Example 4.7. We choose an arbitrary fundamental circuit from the graph G in Figure 4.8, for example, $C_1 = \langle b, c, e, f, g \rangle$.

Let $S \subset G$ be the graph defined by the vertices and edges corresponding to the fundamental circuit C_1, that is S=($\{b$,c,e,f,$g\}$, $\{(b$,$c)$,$(c$,$e)$,$(e$,$f)$,$(f$,$g)$,$(g$,$b)\}$). S induces on G a set of components. According to definition of Chapter 2.7, the components H_2, H_3 of G induced by S are singular components. Component H_1 is a non singular component. The components of G are the following:

$$\begin{aligned} H_1 &= (\{a, b, c, d\}, \{(a, b), (a, c), (a, d)\}) \\ H_2 &= (\{b, f\}, \{(b, f)\}) \\ H_3 &= (\{e, g\}, \{(e, g)\}) \end{aligned}$$

Thus, H_1, H_2, and H_3 are also the bridges of G induced by C_1. Figure 4.10(b) shows the induced bridges in graph in Figure 4.10(a).

4.4.6 Computing the Collapsed Graph

In this section we overview the algorithm to compute a collapsed graph. Roughly, the strategy is to divide the graphs G into a circuit C and the collection B' of collapsed star bridges induced

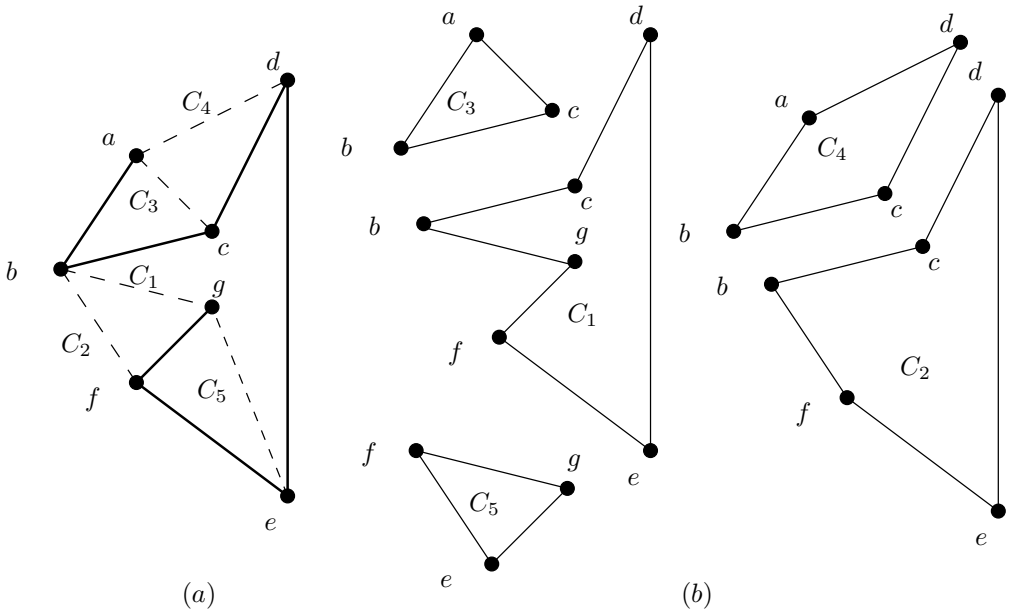


Figure 4.9: (a) Spanning tree of the case study graph and (b) Associated circuits.

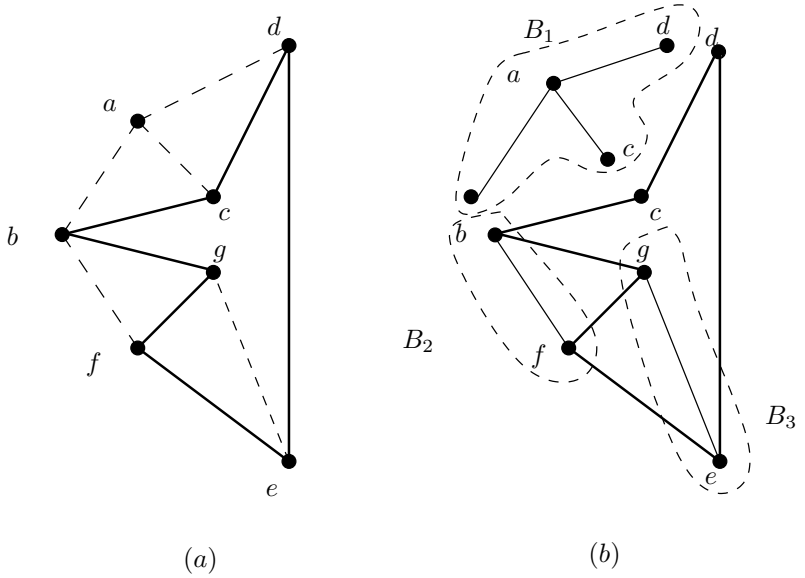


Figure 4.10: (a) Circuit C_1 of the case study graph and (b) Induced bridges B_1, B_2, B_3.

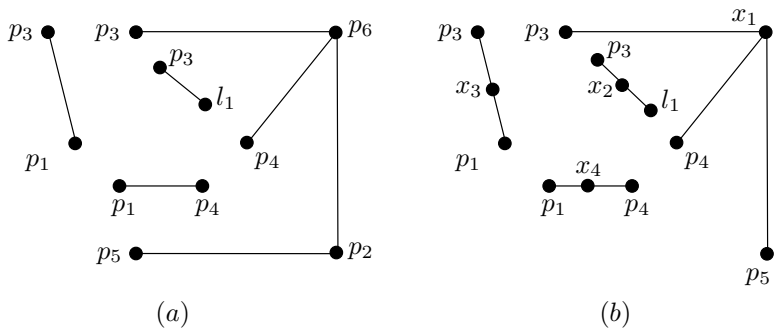


Figure 4.11: (a) Bridges. (b) Star Bridges.

by C in G. For every bridge B induced by C in G, there is a corresponding star graph B' resulting from collapsing the internal vertices of each bridge B to a single vertex.

Consider a graph $G = (V, E)$ and a fundamental circuit $C \subset V(G)$ that induces in G the set of bridges $B = \{B_1, \ldots, B_m\}$. Let $B_i = (V_i, E_i)$ be a bridge in B such that $at(B_i) = \{a_1, \ldots, a_n\} \subseteq V_i$. We define the star graph $S_i = (V, E)$ associated to bridge B_i as the graph such that $V(S_i) = \{x, a_1, \ldots, a_n\}$, where x is a new vertex and, $E(S_i) = \{(x, a_j), 1 \leq j \leq n\}$. The *collapsed graph* of G is the graph resulting from replacing each bridge $B_i \in B$ with the corresponding star bridge $s(B_i)$.

Example 4.8. To illustrate this concept, consider the bridges shown in Figure 4.11(a). The corresponding star bridges are shown in Figure 4.11(b).

Example 4.9. Consider now the graph shown in Figure 4.12(a) with $C = \langle p_1, l_1, p_4, p_5, p_3, l_2 \rangle$. The collapsed graph resulting by replacing every bridge with its star bridge S_i is shown in Figure 4.12(b).

Example 4.10. Finally, we illustrate the procedure applied to the case study graph. The non-singular bridge B_1 and the singular bridges B_2 and B_3 in Figure 4.13(a) are collapsed in Figure 4.13(b).

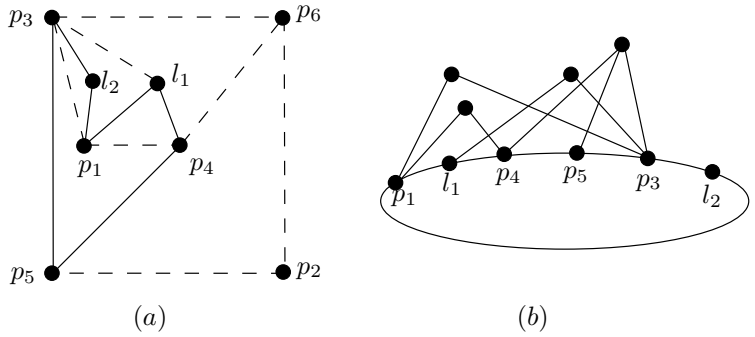


Figure 4.12: (a) Graph (b) Collapsed graph.

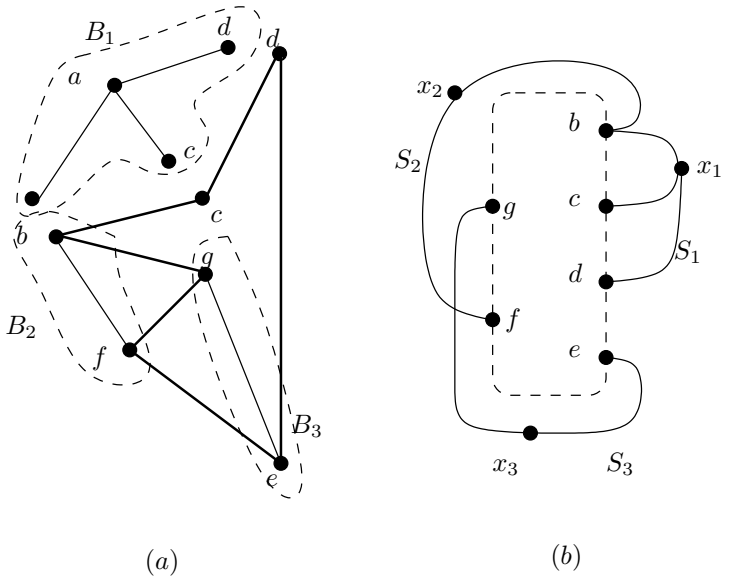


Figure 4.13: (a) Bridges (b) Collapsed Graph.

4.4.7 Computing the Merged Graph

We will further simplify the collapsed graph by merging sets of star bridges that interlace into larger star bridges. Now follows the concept of *merged graph*.

Recall that given fundamental circuit C under consideration and given a bridge B with attachments $at(B) = \{a_1, a_2, \ldots, a_m\} \subset V(C)$, vertices in $at(B_i)$ are sorted according to the circular order in C. Let G' be the collapsed graph resulting from replacing the bridges induced by the fundamental circuit C in the graph $G = (V, E)$ with the associated star graphs. Let S_i and S_j be two stars in G' attached to C. Following Chapter 2.8, we say that stars S_i and S_j interlace if one of the following two holds,

1. There are four distinct vertices a, b, c, d in increasing order in C such that either a, c belong to $at(S_i)$ and b, d belong to $at(S_j)$ or a, c belong to $at(S_j)$ and b, d belong to $at(S_i)$; or

2. $at(S_i)$ and $at(S_j)$ share three distinct vertices.

Example 4.11. The bridges of the graph G induced by circuit $C = \langle a, d, c, k, b\rangle$ in Figure 4.14(a) are $B_1 = \{d, c, b\}$, $B_2 = \{d, a, c, b\}$ and $B_3 = \{d, a\}$. Bridges are represented as star graphs B'_1, B'_2 and B'_3. Figure 4.14(b) shows the collapsed graph. $B'_1 = \{d, c, b\}$ and $B'_2 = \{d, a, c, b\}$ interlace because they have three common attachments.

Notice that star graphs S_i and S_j in the collapsed graph interlace if and only if bridges B_i, B_j interlace in graph G.

Example 4.12. Figure 4.15 shows a collapsed graph G and two interlacing bridges S_1 and S_2. S_1, S_2 interlace due to the fact they have three common attachments.

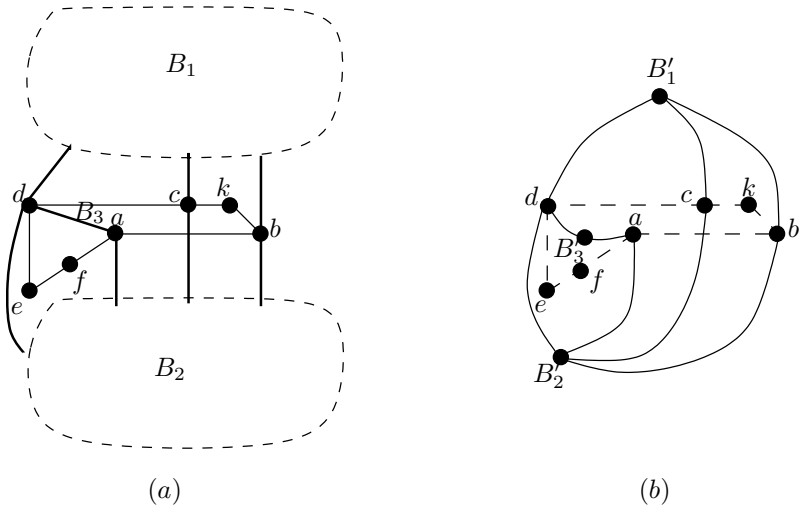


Figure 4.14: (a) Graph and Bridges B_1,B_2,B_3 and (b) Collapsed graph with B'_1 and B'_2 sharing 3 attachments.

Example 4.13. Figure 4.16(b) shows the collapsed graph corresponding to bridges of a graph G. In this example, the analyzed circuit is $C = \langle d, a, c, b, f, g, h\rangle$, and the bridges are $B_1 = \{a, b\}$, $B_2 = \{d, c\}$ $B_3 = \{b, g\}$ and $B_4 = \{f, h\}$. B'_1 and B'_2 interlace because their attachments appear in the circuit in the sequence d, a, c, b. B'_3 and B'_4 interlace because their attachments appear in the circuit in the sequence b, f, g, h.

Next follows two collapsed graphs with no interlacing star bridges.

Example 4.14. See Figure 4.17. Let G be a collapsed graph and let S_1 and S_2 two bridges. S_1 and S_2 do not interlace.

Example 4.15. Figure 4.18 shows a collapsed graph G and two bridges S_1 and S_2. S_1 and S_2 do not interlace.

Let S_i and S_j be two star graphs with centers x_i and x_j respectively and $x_i \neq x_j$. By *merging* stars S_i and S_j we mean replacing these stars with a new one S_{ij} formed by combining the

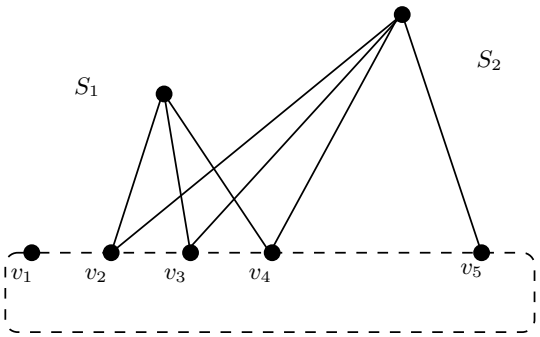


Figure 4.15: Collapsed graph example with stars S_1 and S_2 that interlace.

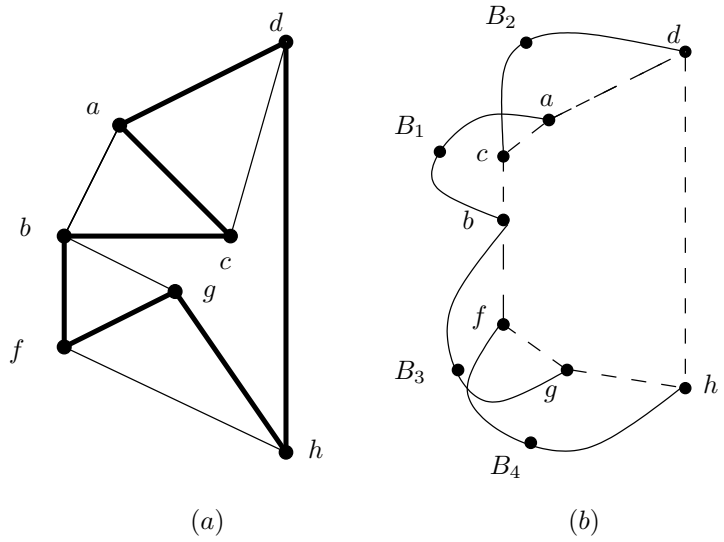


Figure 4.16: (a) Graph and (b) Collapsed graph with several interlacing stars.

attachments of S_i and S_j and identifying the centers. Given a collapsed graph G', the *merged graph* G'' is obtained from G' by merging all pairs of stars that interlace.

Example 4.16. Following the case study graph, Figure 4.19(b) shows the merged graph of the collapsed graph in Figure 4.19(a). S_2 and S_3 interlace.

Since no two star bridges interlace in a merged graph, the merged graph always has a planar embedding. Note that the merged graph is planar. Recall Theorem 2.6.

4.4.8 Computing the Planar Embedding of the Merged Graph

The next step is to compute a planar embedding for the planar merged graph G'' induced in the input graph G by a fundamental circuit.

Many general algorithms to compute planar embedding of planar graphs have been published. See, for example, [95,96]. However, having in mind the properties of the merged graph and the

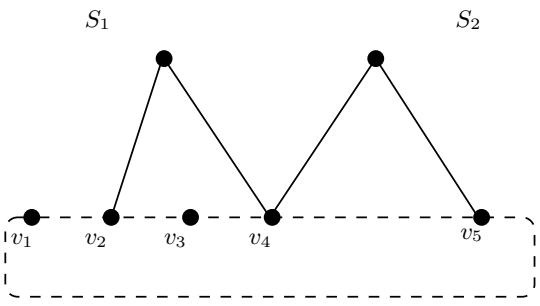


Figure 4.17: Collapsed graph example with stars S_1 and S_2 that do not interlace.

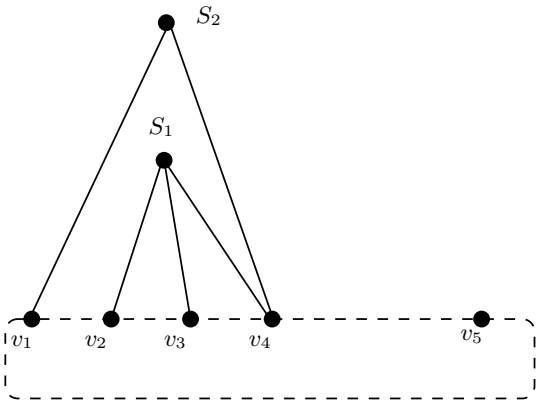


Figure 4.18: Collapsed graph example with stars S_1 and S_2 that do not interlace.

specificity of the embedding we are looking for, we have developed an *ad hoc* simple algorithm.

Following Chapter 2.6, a planar graph can be drawn in the two-dimensional space with no two of its edges crossing. Such a drawing of a planar graph is called a *plane drawing*. Any plane drawing separates the plane into distinct regions bordered by graph edges called *faces*. We want our starting embedding to be defined by embedding the fundamental circuit C under consideration. In these conditions and according to the Jordan Curve Theorem, [24], the resulting embedding has two faces, one is bounded, f_b , and the other is unbounded, f_u. See Figure 4.20. Now, with the aim of speeding up the search for the hinges, we want the bounded face f_b to be preserved all along the computation. Thus, from a computational point of view, face f_b is not included in the embedding representation.

Let C be the circuit considered in the constraint graph and B be the induced merged graph. We start the embedding of G by trivially embedding C. We assume that circuit vertices are circularly labeled with increasing label values and that vertices in the bridges attachments are accordingly sorted. Moreover, we assume that if the circuit under consideration is $C = \langle v_1, v_2, \ldots, v_n \rangle$, the attachments of each bridge $at(B_i) = \{v_{i1}, v_{i2}, \ldots, v_{im}\}$ fulfill $v_1 \leq v_{il}$ and $v_{im} \leq v_n$. See Figure 4.21. Notice that this assumption entails, at most, a coherent relabeling of vertices in the circuit and bridges. Then, we proceed to compute the embedding of the merged graph G'' by recursively adding each of the non-interlacing star bridges belonging to G'.

Example 4.17. See for example the planar embedding in Figure 4.21.

Example 4.18. For the case study graph in Figure 4.8, the planar embedding of the merged graph in Figure 4.19(b) is shown in Figure 4.22. The set of faces is $\{f_u, f_b, f_1, \ldots, f_5\}$.

4.4.9 Computing the Hinges

Let $G = (V, E)$ be the constraint graph, let C be a fundamental circuit in G. Let G' be the corresponding merged graph and F the set of faces of a planar embedding of G'. We will show later on Chapter 5 that if there are two faces $f_i, f_j \in F$ such that $f_i \cap f_j = \{v_a, v_b, v_c\} \subset V(G)$ then $\{v_a, v_b, v_c\}$ are hinges that define a set partition of $V(G)$ and therefore decompose G

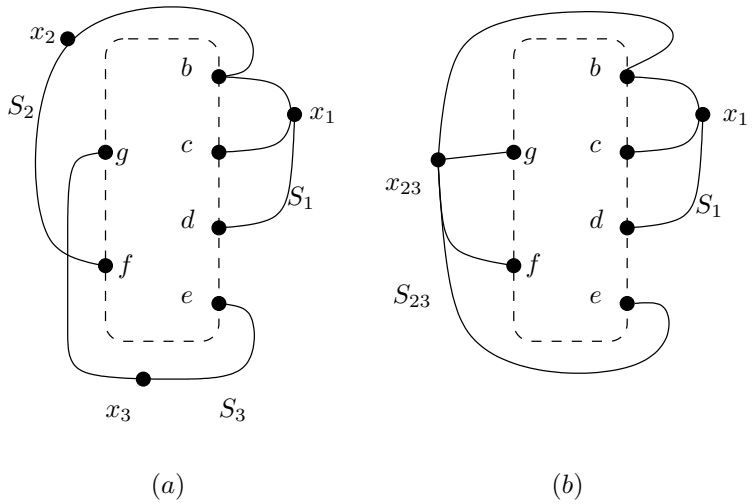


Figure 4.19: (a) Collapsed graph. (b) Merged graph.

into three clusters. In our algorithm we fix the bounded face f_b defined by the circuit under consideration, as one of the two faces to be checked.

Example 4.19. In the planar embedding for the case study shown in Figure 4.22, the set of faces is is $\{f_1, f_2, f_3, f_4, f_5, f_u, f_b\}$. We can see the faces f_b and f_3 share vertices $\{b, d, e\}$, therefore they are hinges that define in $V(G)$ the set partition $V_1 = \{d, e\}$, $V_3 = \{a, b, c, d\}$ and $V_2 = \{b, f, g, e\}$ which induce the graph decomposition into clusters G_1, G_2, G_3 shown in Figure 4.23.

When at given decomposition step two or more possible different decompositions exist, our algorithm does not control, so far, which one is chosen. As a matter of fact the specific decomposition for tree-decomposition based DR-planners are canonical, [34, 60]. Thus, if a set of triple vertices is identified, almost two different decompositions can be performed. In fact, if there are more than three shared vertices, the problem is underconstrained and any subset of three shared vertices are hinges.

4.4.10 Tree Decomposition

Next the algorithm is recursively applied to each of the clusters until reaching trivial geometry placing problem and no further decomposition is needed.

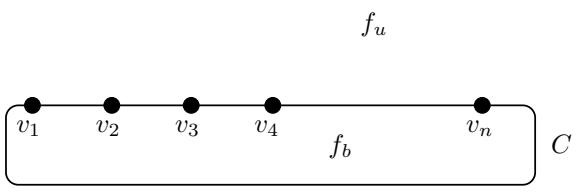


Figure 4.20: A circuit embedded in the plane. Face f_u is unbounded. Face f_b is bounded.

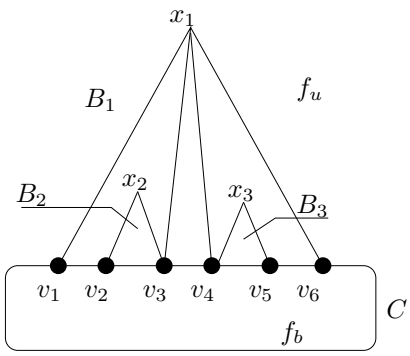


Figure 4.21: Labeling of circuits and bridges.

Example 4.20. Finally, Figure 4.24 shows a collection of set decompositions recursively generated for the tree decomposable graph of Figure 4.8 and the corresponding tree decomposition.

4.5 Case study

To illustrate how the new DR-planner algorithm works, we now show a complete case study in which the algorithm obtains a treedecomposition to a more complex geometric constraint problem. We use a mechanical engineering structure example, *a truss*, [73]. A truss is one of the major types of engineering structures which provides a practical and economical solution for many engineering constructions, specially in the design of bridges and buildings that demand large spans. A truss is a structure composed of slender members joined together at their end points. A simple warren truss bridge is one of the most popular bridge designs. For example, the Continuous Warren box truss bridge with no verticals over Cooper River on I-526 in Charleston. See Figure 4.25, taken from Google Maps.

In our case of study, we take as an example a compound Warren truss bridge, where the bars of a large simple truss, called the main truss, have been substituted by simple truss, called secondary trusses.

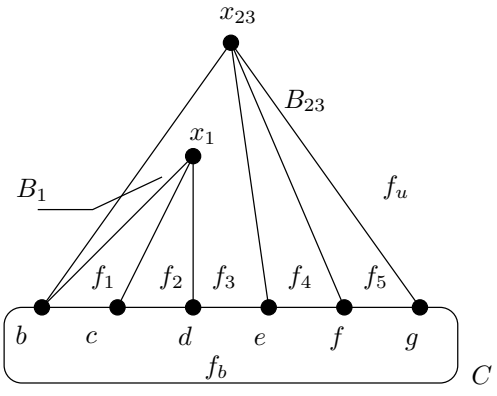


Figure 4.22: Planar embedding for the merged graph given in Figure 4.19b).

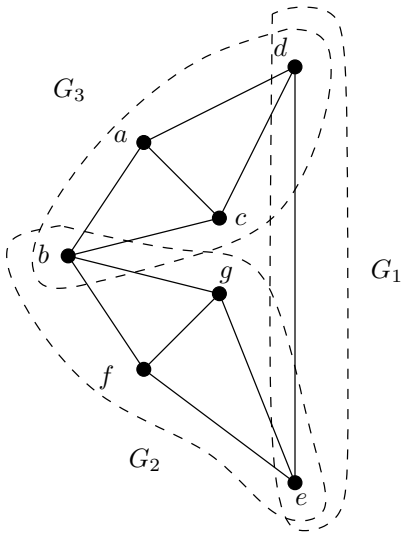


Figure 4.23: Graph and clusters G_1, G_2 and G_3.

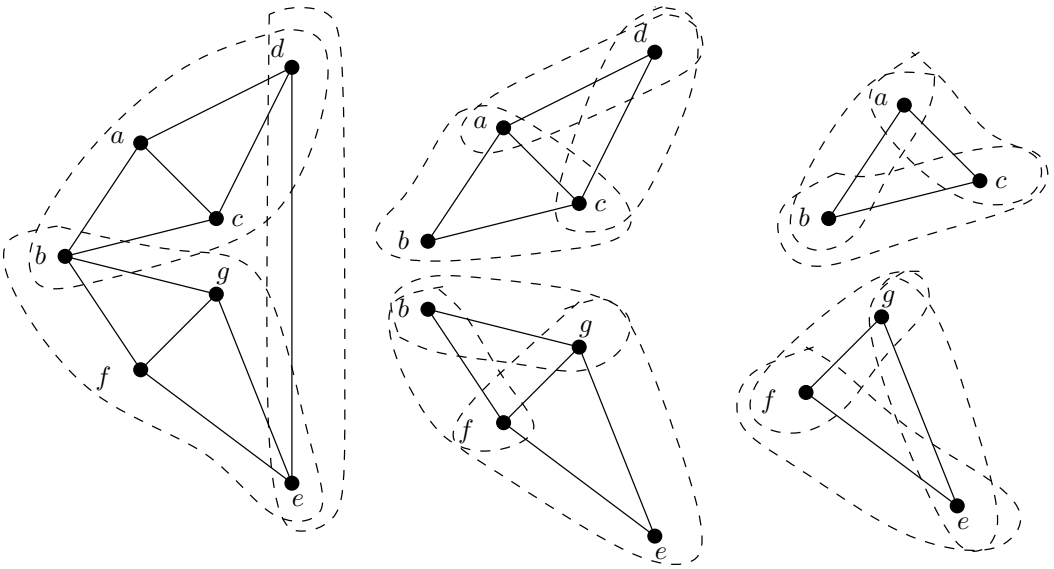
Figure 4.26 shows the geometric constraint problem and Figure 4.27(a) shows the corresponding constraint graph $G = (V, E)$ of study. Vertices $V(G)$ are the truss joints and bars connecting two joints define the set of edges $E(V)$. Here constraints are distances between joints.

4.5.1 Computing the Set of Fundamental Circuits

First, we apply a depth-first search to graph G, starting in the arbitrary root a. We obtain the tree shown in Figure 4.27(b).

The chords are the edges of Figure 4.27(b), not visited in the previous step. Every chord induces a fundamental circuit. Figure 4.28 shows the resulting DFS tree and the fundamental circuits (chords) $C_1, C_2, \ldots, C_9$. The fundamental circuits contain the following vertices:

$$
\begin{aligned}
C_1 &= \langle a, f, g, b, k, j, c, i, h, d, e\rangle \\
C_2 &= \langle a, f, g\rangle \\
C_3 &= \langle f, g, b\rangle \\
C_4 &= \langle k, j, c\rangle \\
C_5 &= \langle b, k, j, c, i, h, d, e\rangle \\
C_6 &= \langle c, i, h, d, e\rangle \\
C_7 &= \langle b, k, j, c, i\rangle \\
C_8 &= \langle c, i, h\rangle \\
C_9 &= \langle i, h, d\rangle
\end{aligned}
$$



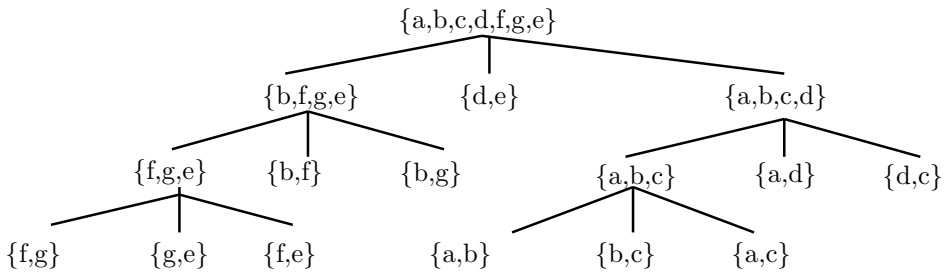


Figure 4.24: Collection of set decompositions for the case study graph in Figure 4.8.

Figure 4.25: Warren truss bridge over Cooper River in Charleston - South Carolina (Source: Google Maps).

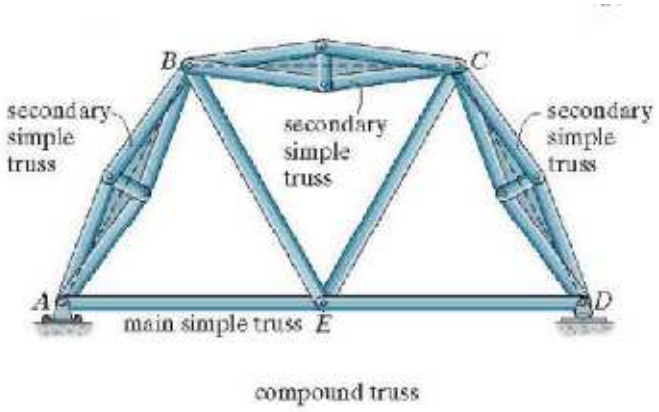


Figure 4.26: Bridge coplanar compound truss.

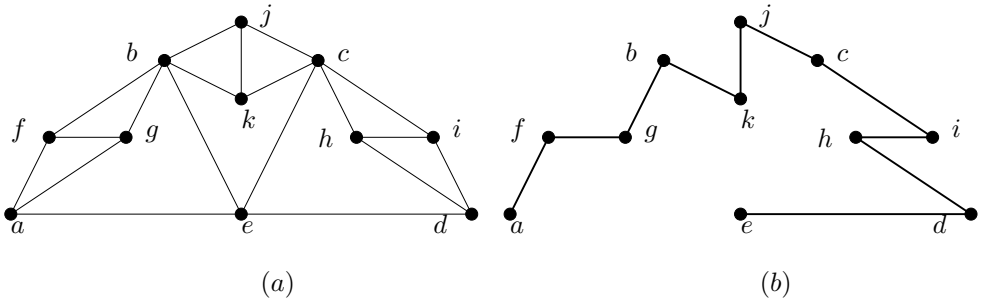


Figure 4.27: (a) Truss constraint graph (b) Spannig tree.

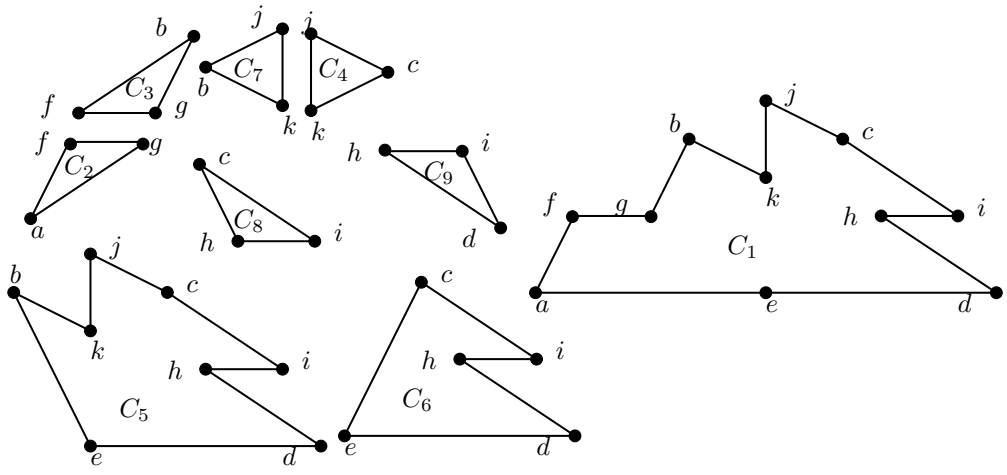


Figure 4.28: Fundamental circuits in the truss graph associated to the spanning tree depicted in Figure 4.27(b).

4.5.2 Computing the Set of Bridges

We proceed to explore circuits to search hinges. We choose an arbitrary fundamental circuit from the graph G, for example: $C_5 = \langle b, k, j, c, i, h, d, e \rangle$, see Figure 4.28.

Let $S \subset G$ be the graph defined by the vertices and edges corresponding to the fundamental circuit C_5, that is $S = (\{b, k, j, c, i, h, d, e\}, \{(b, k), (k, j), (j, c), (c, i), (i, h), (h, d), (d, e), (e, b)\})$. The dashed circle in Figure 4.29 shows the class κ_1 defined by the circuit C_5, depicted in thick line, in the running truss graph example.

S induces on G a set of components. There is one non-singular component and the rest are singular components. The set of components is shown in Figure 4.30.

The set of bridges in the truss graph corresponding to the components in Figure 4.30 is depicted in Figure 4.31 and are the following:

$$
\begin{aligned}
B_1 &= (\{a, f, g, b\}, \{(a, f), (a, g), (f, g), (f, b), (g, b)\}) \\
B_2 &= (\{b, j\}, \{(b, j)\}) \\
B_3 &= (\{k, c\}, \{(k, c)\}) \\
B_4 &= (\{c, e\}, \{(c, e)\}) \\
B_5 &= (\{c, h\}, \{(c, h)\}) \\
B_6 &= (\{i, d\}, \{(i, d)\})
\end{aligned}
$$

4.5.3 Computing the Collapsed graph

Figure 4.32(a) shows the collapsed graph G' corresponding to graph G after replacing every bridge H_i by a star B_i, and where each bridge is identified by its center vertex.

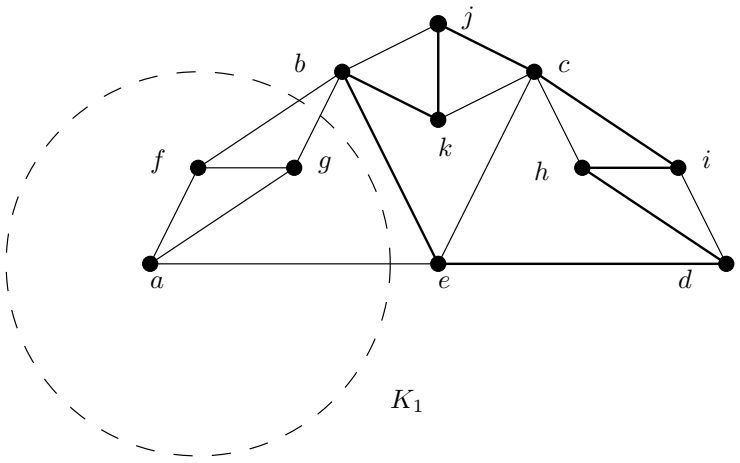


Figure 4.29: Classes induced by the circuit C_5, (thick line) in the truss graph.

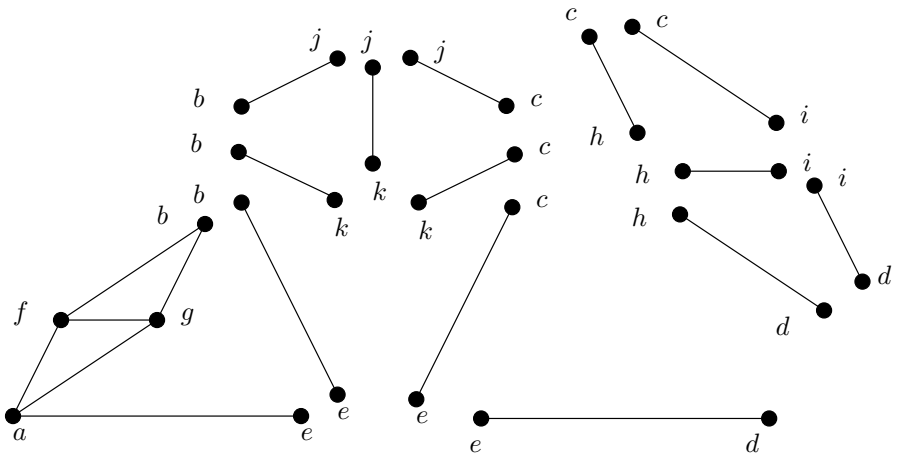


Figure 4.30: Components defined by the classes in the truss graph for the circuit C_5.

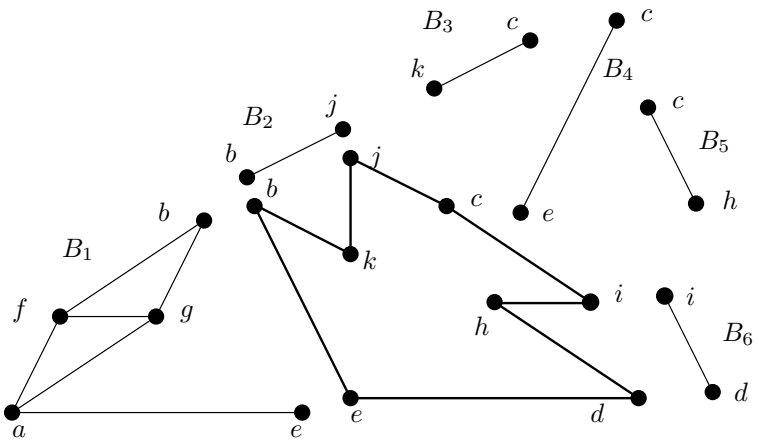


Figure 4.31: Bridges (thin line) induced in the truss graph by circuit C_5.

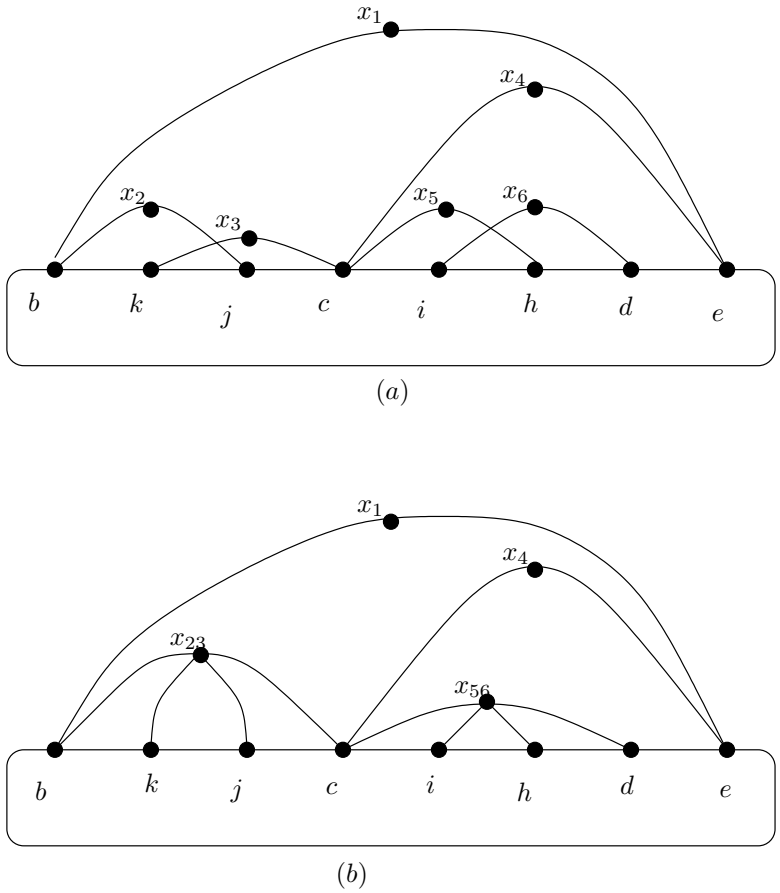


Figure 4.32: Truss graph example. (a) Collapsed Graph (b) Merged Graph.

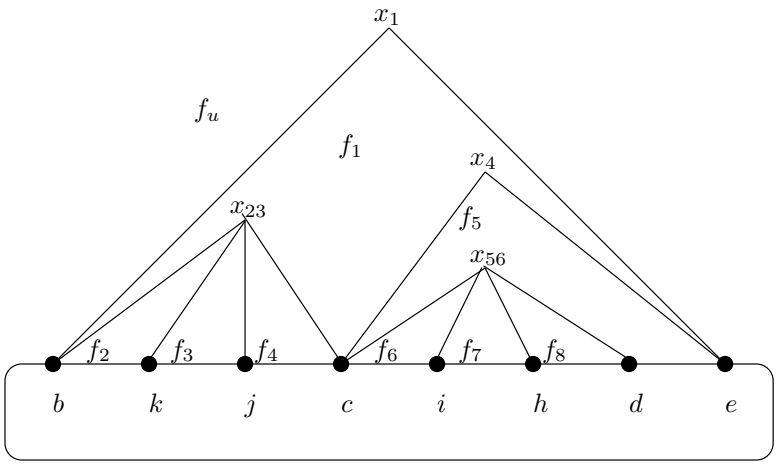


Figure 4.33: Planar embedding for the running merged truss graph.

4.5.4 Computing the Merged Graph

We proceed to merge bridges of G'. Bridges B_2 and B_3 have interlaced attachments and are merged into B_{23}. In a similar way, bridges B_5 and B_6 interlace and are merged into B_{56}. The resulting merged graph of G', named G'', is shown in Figure 4.32(b).

4.5.5 Computing the Planar Embedding of the Merged Graph

Next step is to compute a planar graph G'''. Figure 4.33 shows the planar graph G'''. This is the result of the planar embedding of the merged graph G''.

The set of faces defined by the planar embedding of G''' are the following.

$$
\begin{aligned}
f_u &= \{b, x_1, e\} & f_1 &= \{b, x_{23}, c, x_4, e, x_1\} \\
f_2 &= \{b, k, x_{23}\} & f_3 &= \{k, j, x_{23}\} \\
f_4 &= \{j, c, x_{23}\} & f_5 &= \{c, x_{56}, d, e, x_4\} \\
f_6 &= \{c, i, x_{56}\} & f_7 &= \{i, h, x_{56}\} \\
f_8 &= \{h, d, x_{56}\}
\end{aligned}
$$

4.5.6 Computing the Hinges

To compute a set of hinges, we search the set of faces $\{f_u, f_1, \ldots, f_8\}$. The faces f_1 and f_5 contains three vertices of the circuit C_5. Therefore, the sets $\{b, c, e\}$, and $\{c, d, e\}$ are hinges. For example, for the set $\{b, c, e\}$, the induced graph decomposition into clusters G_1, G_2, G_3 is shown in Figure 4.34.

4.5.7 Recursive Tree Decomposition Algorithm

Figure 4.35 shows one recursive tree decomposition for G.

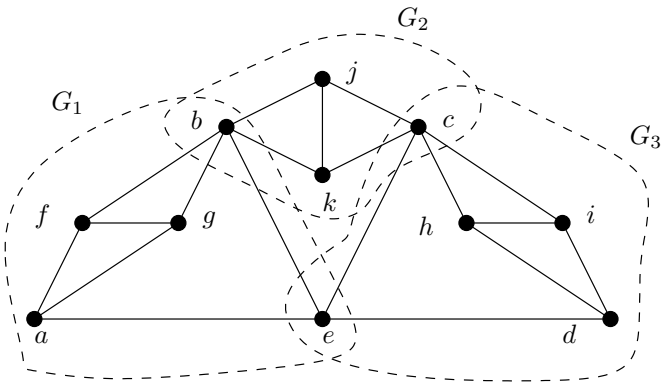


Figure 4.34: Graph partition for the running graph defined by hinges $\{b, c, e\}$ and induced clusters G_1, G_2 and G_3.

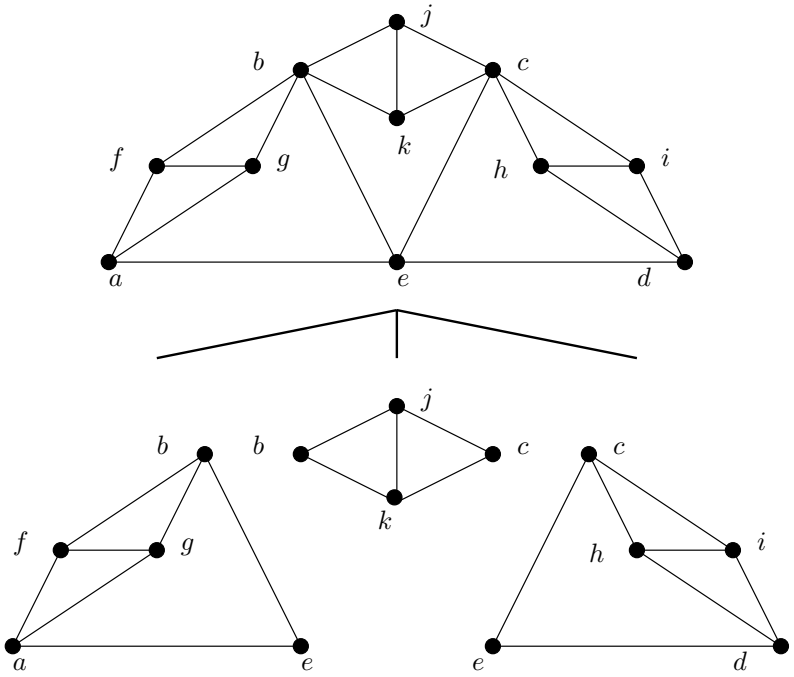


Figure 4.35: Recursive treedecomposition of the truss graph G.

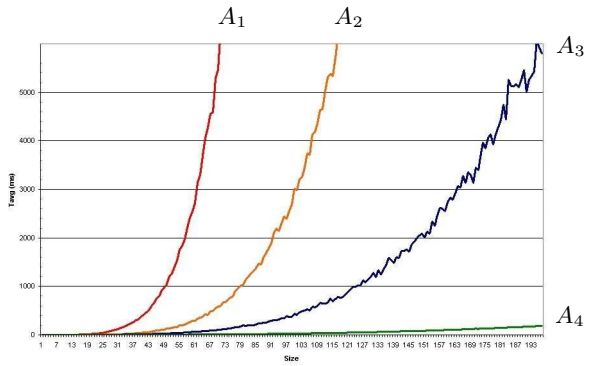


Figure 4.36: Behavior of the algorithms A_1, A_2, A_3, and A_4 on the dataset D_1.

4.6 Experimental Runtime Behavior

To gain insight on the algorithm behavior and to perform a preliminary assessment of the algorithm runtime behavior, we implemented it in the `SolBCN` framework which can be downloaded under a GNU General Public License (see [114]). Using the methodology defined in Joan-Arinyo *et al.*, [64], the tests were conducted on a standard desk computer with a Pentium IV at 3GHz processor and 1GB of core memory and two datasets were defined and planned as follows.

1. D_1: A set of 1000 randomly generated geometric constraint graphs with sizes ranging from 3 to 200 vertices. All the graphs were well-constrained but not necessarily tree-decomposable, that is not necessarily solvable by the tree decomposition approach.

2. D_2: A set of 1000 randomly generated of geometric constraint graphs with sizes ranging from 3 to 200 vertices. All the graphs were under-constrained but not necessarily tree-decomposable.

We also defined four versions of the decomposition algorithm:

1. A_1: this is a *brute-force* algorithm that performs an exhaustive search for hinges.

2. A_2: In this version first vertices of degree two are removed. Then the brute-force algorithm is applied.

3. A_3: This version is an improvement of A_2 with specific treatment for 0-connected and 1-connected graphs.

4. A_4: is the algorithm presented in this work.

Let SG_s denote the set of graphs G such that $s = |V(G)|$. Notice that $3 \leq s \leq 200$. We applied each algorithm version to each dataset. For each algorithm and each graph in a data

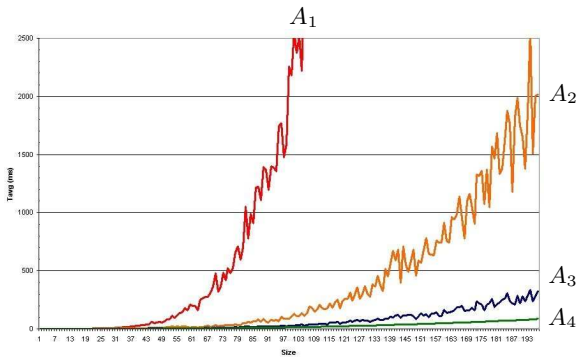


Figure 4.37: Behavior of the algorithms A_1, A_2, A_3, and A_4 on the dataset D_2.

set, we recorded the algorithm runtime $t(G)$. Then for each graph size s, we averaged the runtime values as

$$T(s) = \frac{\sum_{\forall G \in SG_s} t(G)}{s}$$

The results yielded by these tests are represented in Figure 4.36 for dataset D_1 and in Figure 4.37 for dataset D_2.

The new DR-planner algorithm was tested and the empirical behavior was documented in [70], and show that for both datasets the algorithm A_4 introduced in this paper exhibits a noticeable improved behavior. For graphs G with $|V(G)| \approx 200$, the runtime for the algorithm A_4 is of about 200ms what allows interactive use in the SolBCN framework.

4.7 Chapter Summary

We have described a new decomposition-recombination planning method for geometric constraint solving based on tree-decomposition. Compared to other decomposition-recombination methods, the tree-decomposition based DR-planner algorithm proceeds in a clear and simple way to recursively split the input graph of geometric constraints to three subgraphs such that pairwise share one vertex. We distinguish between 0-connected, 1-connected and biconnected input graphs. For the case of 0-connected and 1-connected, a simple approach allows to find the hinges that decompose the input graph into three subgraphs.

For the case of biconnected graphs, hinges are found in a particular embedding of the graph considering one fundamental circuit after a set of transformations. The search for hinges in the embedding is done by fixing one of the faces of the embedding, the bounded face f_b, and checking whether there is a face in the embedding which shares three vertices with face f_b. If such a face exists, the common vertices are a triple of hinges that split the input graph into three subgraphs. If the search for hinges fails, a new fundamental circuit is selected and the process is applied again. If after visiting all the fundamental circuits no triple of hinges is found, the input graph is not tree-decomposable. Once the hinges are found, the input graph is split into three subgraphs and the process is recursively applied to each subgraph until the resulting subgraphs are trivial.

Sections 1 through 3 in Appendix 7 include case studies to further illustrate our approach.

CHAPTER 5

Correctness Proof

Honesty's the best policy.

(Miguel de Cervantes)

In this chapter we prove that the new tree-decomposition algorithm for the DR-planner, presented in Chapter 4 is correct in the sense that if a graph is decomposable, the algorithm computes a set of hinges.

We present correctness-preserving transformations that can be applied to the geometric constraint graph in the presented tree-decomposition DR-planner algorithm. We determine first that every set of hinges in the graph G is contained in one of the fundamental circuits of G. Then we show that during the conversion of the input graph $G = (V, E)$ to a collapsed graph and finally to a merged graph, the hinges remain unaltered. Next we demonstrate that the resulting merged graph is planar. Finally we prove the main result, a merged graph is tree decomposable if and only if there is a planar embedding with two faces which share three or more vertices.

Most of the work of this chapter has been previously published in reference [71].

5.1 Hinges and Fundamental Circuits

The strategy of the algorithm introduced in Chapter 4 relies on the fact that given a spanning tree for a biconnected, tree decomposable constraint graph, every set of hinges is always included in some fundamental circuit of the graph. Thus, we go ahead analyzing the relation between hinges and fundamental circuits in biconnected graphs. We prove that if a graph G is decomposable, there is always a fundamental circuit of G containing a set of hinges.

In what follows, if $\{a, b, c\}$ are hinges to a biconnected decomposable graph $G = (V, E)$ we will denote by G_{ab}, G_{bc} and G_{ca} the graphs resulting from the set decomposition of G such that $V(G_{ab}) \cap V(G_{bc}) = b$, $V(G_{bc}) \cap V(G_{ca}) = c$ and $V(G_{ca}) \cap V(G_{ab}) = a$.

We start by proving Lemma 5.1,

Lemma 5.1. *Let $G = (V, E)$ be a biconnected, tree decomposable graph with hinges $\{a, b, c\}$ and clusters G_{ab}, G_{bc} and G_{ca}. Let C be a circuit in G. Then one of the following two holds*

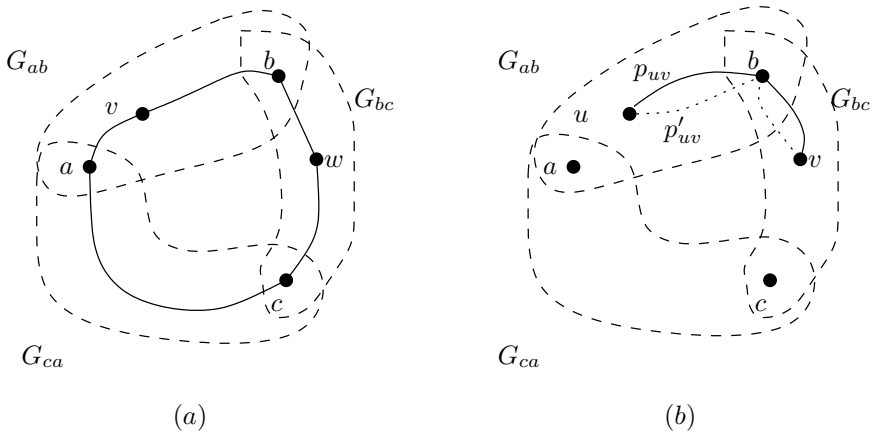


Figure 5.1: Circuits in a tree decomposition of a biconnected graph. (a) The circuit shares vertices with the three subgraphs in the set decomposition. (b) A circuit cannot intersect just two subgraphs in a set decomposition.

1. *Circuit C shares vertices with clusters G_{ab}, G_{bc} and G_{ca}. Furthermore $\{a, b, c\} \subseteq V(C)$.*

2. *$V(C)$ is a subset of either $V(G_{ab})$ or $V(G_{bc})$ or $V(G_{ca})$.*

Proof. For the first case just apply the definition of tree decomposition of a graph. See Figure 5.1(a).

For the second case assume, without loss of generality, that C is a circuit of G such that it shares vertices with $V(G_{ab})$ and $V(G_{bc})$ but it has no vertices in common with $V(G_{ca})$. See Figure 5.1(b). For a contradiction let u, v be two vertices in $V(C)$, $u \neq v$, such that $u \in V(G_{ab})$ and $v \in V(G_{bc})$. Since C is a circuit, there are two distinct paths in C, say p_{uv} and p'_{uv}, such that connect vertices u and v. Both paths must include the hinge $\{b\} = V(G_{ab}) \cap V(G_{bc})$. Therefore C is not a circuit. □

Theorem 5.1 states that if a circuit C shares vertices with clusters G_{ab}, G_{bc} and G_{ca}, then C is a fundamental circuit of G.

Theorem 5.1. *Let $G = (V, E)$ be a biconnected tree decomposable graph with hinges $\{a, b, c\} \in V$ and let T be a spanning tree to G. Then, there is a fundamental circuit, say C, such that $\{a, b, c\} \subseteq V(C)$.*

Proof. Assume that $G = (V, E)$ is a tree decomposable, biconnected graph with hinges $\{a, b, c\}$ that split G into clusters G_{ab}, G_{bc} and G_{ca}. Let $\{C_1, \ldots, C_n\}$ be the set of fundamental circuits of G with respect to a spanning tree T.

For a contradiction assume that there is no fundamental circuit $C_i \in \{C_1, \ldots, C_n\}$ such that $\{a, b\} \subseteq V(C_i) \cap V(G_{ab})$ and $\{b, c\} \subseteq V(C_i) \cap V(G_{cb})$ and $\{c, a\} \subseteq V(C_i) \cap V(G_{ca})$. Then, according to Lemma 5.1, each fundamental circuit C_i is a subset of either G_{ab} or G_{bc} or G_{ca}. Consequently, the ring sum of fundamental circuits in cluster G_{xy} results in circuits which are subgraphs of cluster G_{xy}.

Since G is biconnected, there are two vertex disjoint paths that connect any pair of vertices in $V(G)$. Thus there is a circuit, say C', such that connects the three hinges $\{a, b, c\}$, has non

empty intersection with clusters G_{ab}, G_{bc} and G_{ca} and, by hypothesis is not a fundamental circuit. Therefore, C' can be expressed as the ring sum of some fundamental circuits of G induced by the spanning tree T. This fact contradicts the assumption. □

From this we may conclude that given a graph G and the set of all the fundamental circuits $S = \{C_1, \ldots, C_m\}$ of G with respect to a spanning tree T, there is a circuit $C \in S$ that contains the three hinges $\{a, b, c\}$.

Therefore the decomposition algorithms starts by computing a spanning tree T and the set of fundamental circuits C induced in the graph $G = (V, E)$ by T.

5.2 The Algorithm Preserves Hinges

As explained in Chapter 4, the algorithm iterates over the set of fundamental circuits already computed searching for a planar embedding with two faces sharing three vertices in $V(G)$. For each fundamental circuit, the decomposition algorithm computes the bridges, the collapsed graph and the merged graph. Here we prove that in all those steps the algorithm preserve the hinges of the original graph.

Definition 5.1. Assume that $G = (V, E)$ is a biconnected, tree decomposable planar graph with hinges $\{a, b, c\}$ on the fundamental circuit $C \subset V$. Hinges define in C three *circuit segments*, say C_{ab}, C_{bc} and C_{ca} where each C_{uv} with $u, v \in \{a, b, c\}$, defines a path that connects vertices u and v along the circuit C which does not include the third hinge. See Figure 5.2. Note that here and in the following, when we refer to a circuit segment C_{uv}, with $u, v \in \{a, b, c\}$, we assume that $u \neq v$ and that C_{uv} and C_{vu} are the same circuit segment.

Recall that, as described in Chapter 4, bridges are the components induced in G by $V(C)$, after a precise detailed step. We first prove by Lemma 5.2 that each bridge is attached to just one circuit segment induced by the hinges in the fundamental circuit under consideration.

Lemma 5.2. *Let $G = (V, E)$ be a biconnected tree decomposable graph, $C \subseteq G$ be a fundamental circuit and let $B = \{B_1, \ldots, B_k\}$ be the set of bridges induced by C in G. Assume that $\{a, b, c\} \subseteq V(C)$ are three distinct vertices which induce in C the segments C_{ab}, C_{bc} and C_{ca}. Then, $\{a, b, c\}$ are hinges of G if and only if the attachments of each bridge $B_i \in B$ belong to either C_{ab} or C_{bc} or C_{ca}.*

Proof. For the if part assume that $\{a, b, c\}$ are hinges of G, v, w are two different vertices in $at(B_i)$ and there are two different segments of C, say C_{ab} and C_{ca}, such that $v \in C_{ab}$ and $w \in C_{ca}$. See Figure 5.2. Since B_i is connected, there is a path in B_i connecting vertices v and w. Thus $\{a, b, c\}$ are not hinges of G.

For the only if part assume that $\forall B_i \in B$, there are $x, y \in \{a, b, c\}$ such that $at(B_i) \subseteq C_{xy}$. Now we can define the subgraphs $G_{xy} = C_{xy} \cup \{B_i \in B | at(B_i) \subseteq C_{xy}\}$ for $x, y \in \{a, b, c\}$. Note that $\bigcup_{x,y \in \{a,b,c\}} G_{xy}$ and $G_{ab} \cap G_{ca} = a$, $G_{ab} \cap G_{bc} = b$ and $G_{bc} \cap G_{ca} = c$. Thus G_{xy} is a set decomposition of G and $\{a, b, c\}$ is a set of hinges. □

Lemma 5.3 states that collapsing bridges does not change the connectivity of their attachments. The proof that follows first establishes that two bridges interlace only if bridges belong to the same circuit segment. Then, we show that the process of collapsing bridges neither create nor update hinges.

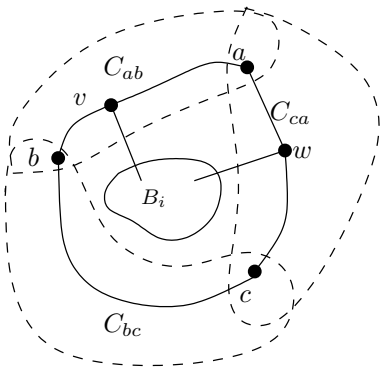


Figure 5.2: C_{ab}, C_{bc} and C_{ca} are circuit segments induced by hinges $\{a, b, c\}$. Bridges share attachments with just one circuit segment induced in the fundamental circuit by the hinges. The illustrated case is not possible.

Lemma 5.3. *Let $G = (V, E)$ be a biconnected, tree decomposable graph, let C be a fundamental circuit of G and let $B = \{B_1, \ldots, B_k\}$ be the set of bridges induced by $V(C)$. Let G' be the collapsed graph resulting from replacing each $B_i \in B$ with the corresponding star bridge. Then, vertices $\{a, b, c\}$ are hinges of G if and only if they are hinges of G'.*

Proof. Just notice that vertices in the circuit C and bridges attachments are not involved in the replacement of a bridge with a star graph. Therefore vertices in C and bridges attachments are neither created nor removed.

According to Lemma 5.2, for each bridge B_i there is a fundamental circuit segment, say C_{uv}, induced in C by hinges $\{a, b, c\}$ of $V(G)$ such that $at(B_i) \subseteq V(C_{uv})$. We consider two different situations depending on whether the bridge B_i is or is not singular. First assume that B_i is not singular. B_i is transformed into a star bridge B_i by merging those vertices which do not belong to $at(B_i)$ into a single vertex. Clearly, the fundamental circuit segment to which the bridge is attached does not change. Assume now that B_i is a singular bridge. The star bridge is computed by breaking the unique bridge edge into two edges and including a new dummy vertex which does not belong to $at(B_i)$. Again, the fundamental circuit segment to which the bridge is attached remains unchanged. □

Next we proceed to prove that computing the merged graph also preserves hinges. Let us start proving by Lemma 5.4 that two bridges in the collapsed graph interlace only if they are attached to the same fundamental circuit segment.

Lemma 5.4. *Let $G = (V, E)$ be a biconnected tree decomposable graph. Let $a, b, c \in V$ be the hinges that define in the fundamental circuit C the circuit segments C_{ab}, C_{bc},C_{ca}. Let $B = \{B_1, \ldots, B_k\}$ be the set of bridges induced by C in G and $s(B) = \{s(B_1), \ldots, s(B_k)\}$ the set of corresponding star bridges. If two star bridges $s(B_i), s(B_j) \in s(B)$ interlace, then there is a circuit segment C_{uv} such that $at(s(B_i)) \subseteq V(C_{uv})$ and $at(s(B_j)) \subseteq V(C_{uv})$.*

Proof. Consider that each bridge B_i is transformed into a star bridge $s(B_i)$ preserving attachments, that is, $at(B_i) = at(s(B_i))$. Now apply Lemma 5.2. □

As shown by Lemma 5.5, in the process of merging bridges in the collapsed graph hinges are preserved.

Lemma 5.5. *Merging star bridges preserves hinges.*

Proof. Assume that the fundamental circuit is C and the hinges are $\{a, b, c\}$. By Lemma 5.4, merging star bridges only merges bridges whose attachments belong to the same fundamental circuit segment either C_{ab}, C_{bc} or C_{ca}. Therefore $\{a, b, c\}$ are hinges for the merged graph. □

Therefore, we have the following result stated in Theorem 5.2.

Theorem 5.2. *Let $G = (V, E)$ be a biconnected tree decomposable graph with $\{a, b, c\}$ vertices on the fundamental circuit C of G. Then $\{a, b, c\}$ are hinges to the collapsed graph and to the merged graph induced by the fundamental circuit C in G if and only if $\{a, b, c\}$ are hinges to the graph $G = (V, E)$.*

Proof. Apply Lemma 5.3 and Lemma 5.5. □

5.3 The Merged Graph is Planar

The goal now is to show that the resulting merged graph is planar.

Lemma 5.6 and Lemma 5.7 show respectively that the merged graph is planar and that each of the clusters of a set decomposition of a planar graph are also planar graphs.

Lemma 5.6. *Let $G = (V, E)$ be a biconnected, tree decomposable graph and let C be a fundamental circuit with hinges $\{a, b, c\}$. Let G' and G'' respectively the collapsed and the merged graphs induced by hinges $\{a, b, c\}$ in C. Then G'' is planar.*

Proof. See Even, [28]. □

Lemma 5.7. *Let G be a biconnected decomposable planar graph with hinges $\{a, b, c\}$ that split G into clusters G_{ab}, G_{bc} and G_{ca}. Then graphs G_{ab}, G_{bc} and G_{ca} are planar.*

Proof. Clearly, subgraphs of a planar graph are planar. See Even, [28]. □

Let $G = (V, E)$ be a planar graph and let D be a planar embedding of G. Each face $f_i \in D$ is bounded by a circuit of G and there is one unbounded face in D, called the external face. Recall that the embedding D is not unique. Now follows Theorem 5.3.

Theorem 5.3. *Let $G = (V, E)$ be a planar graph and let D be an embedding of G. For every face f in D, there exists a planar realization, say D', of the same graph, which has the same set of faces and such that the face F is the external face in D'.*

Proof. See Even, [28]. □

5.4 Decomposability

Now we proceed to prove the correctness of the last step of the decomposition algorithm by Theorem 5.4. Let $G = (V, E)$ be the input graph of the decomposition algorithm and let G' be the corresponding merged graph. The goal of this step is to find one embedding D of G' and two faces $f_i, f_j \in D$ such that share three or more vertices. We will show that if the number of shared vertices is three, they are the hinges of the tree decomposition of G. If there are more than three shared vertices, the problem is underconstrained and any subset of three shared vertices are hinges.

Theorem 5.4. *Let $G = (V, E)$ be a biconnected planar graph. G is tree decomposable with hinges $\{a, b, c\} \subseteq V$ if and only if there is a planar embedding D with two faces $f_i, f_j \in D$ such that $\{a, b, c\} \subseteq boundary(f_i) \cap boundary(f_j)$.*

Proof. We first prove that if $\{a, b, c\}$ are hinges that decompose the graph G, there is a planar embedding D of G with two faces of D whose boundaries share the hinges.

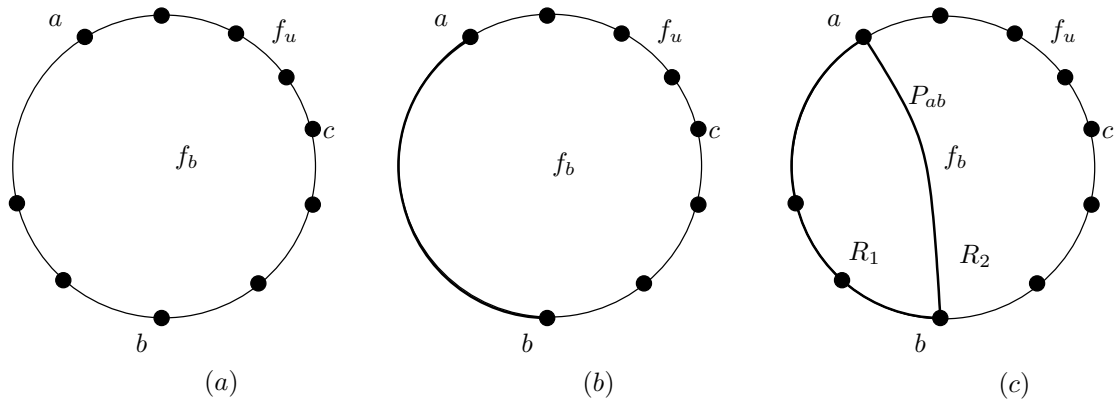


Figure 5.3: (a) Planar embedding of a fundamental circuit with hinges $\{a, b, c\}$. (b) Cluster G'_{ab} with just two vertices. (c) Regions defined by a path in cluster G'_{ab} connecting hinges a and b.

Let $G = (V, E)$ be a biconnected tree decomposable planar graph with hinges $\{a, b, c\}$. Let G' be the graph resulting from collapsing and merging G where hinges $\{a, b, c\}$ induces the clusters G'_{ab}, G'_{bc} and G'_{ca}. Since according to Lemma 5.6 and Lemma 5.7, G' is a planar graph, subgraphs G'_{ab}, G'_{bc} and G'_{ca} are also planar. Now by Theorem 5.1 there is a fundamental circuit C of G' such which includes hinges $\{a, b, c\}$. Since a circuit is planar, C can be embedded in the plane defining two faces. One face is bounded, say f_b and the other face is unbounded, say f_u, as illustrated in Figure 5.3(a). Let us denote this embedding by D.

We build an embedding of G' with all faces drawn within the bounded face f_b of D as follows. Consider the cluster G'_{ab}. Depending on the cardinality of $V(G'_{ab})$, there are two possible different situations. If $|V(G'_{ab})| = 2$, then the cluster is coincident with the fundamental circuit segment C_{ab} and the set of faces in the current embedding D remains unchanged. See Figure 5.3(b).

Assume now that $|V(G'_{ab})| \geq 3$. The cluster G'_{ab} is planar thus it can be embedded within the bounded face f_b in D starting the embedding from vertices and edges in the fundamental

circuit segment C_{ab}. Biconnectivity guarantees that there is another path, say P_{ab}, such that connects vertex a with vertex b, bounds the embedding and splits face f_b into two faces, R_1 and R_2. See Figure 5.3(c).

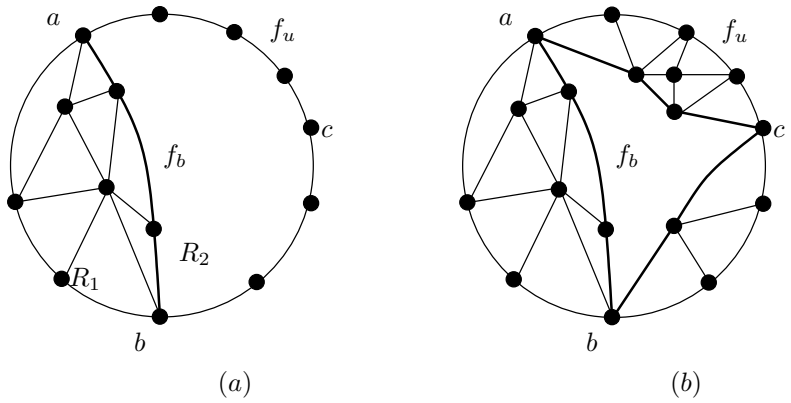


Figure 5.4: (a) Planar embedding of cluster G'_{ab}. (b) Final embedding including clusters G'_{ab}, G'_{bc} and G'_{ca}.

According to Lemma 5.4, bridges included in a cluster are such that $G'_{ab} \cap C_{bc} = \{b\}$, $G'_{ab} \cap C_{ca} = \{a\}$ and $G'_{bc} \cap C_{ca} = \{c\}$. Therefore, by Theorem 5.3, there is an embedding of G'_{ab} within the face labeled R_1 such that the embedding exactly shares vertex a with circuit C_{ca} and vertex b with circuit C_{bc}. See Figure 5.4(a). Similarly, graphs G'_{bc} and G'_{ca} can be embedded in face R_2 resulting in an internal face, labeled f_b in Figure 5.4(b), which always shares with the unbounded face f_u hinges $\{a, b, c\}$. Then, by Theorem 5.2, $\{a, b, c\}$ are hinges for the given graph $G = (V, E)$. The fact that replacing the embedding of each cluster G'_{xy}, say D'_{xy}, with the embedding of the corresponding subgraph $G_{xy} \subset G$ that originated it affects just faces and vertices in D'_{xy}, completes the proof.

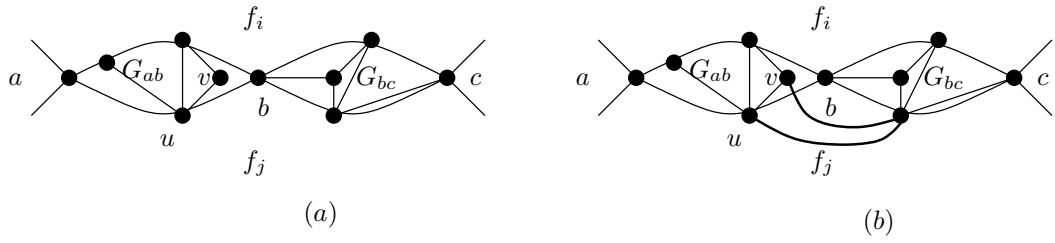


Figure 5.5: Planar embedding where faces f_i and f_j share vertices $\{a, b, c\}$.

For the only part, assume now that there is a planar embedding D of a graph $G = (V, E)$ and two faces $f_i, f_j \in D$ which shares vertices $\{a, b, c\}$, as illustrated in Figure 5.5(a). Then the set of vertices $\{a, b, c\}$ induce a partition of $V(G)$ into three subsets V_{ab}, V_{bc} and V_{ca} such that any path connecting vertex $v \in V_{xy}$ with any vertex z not in V_{xy} must include either vertex x or vertex y. For a contradiction, assume that there is a path between a vertex $v \in V_{ab}$ and a vertex in V_{bc} which does not include the vertex b common to V_{ab} and V_{bc}. There are two different situations illustrate in Figure 5.5(b). Consider first the case where at least one of the

vertices connected by the path does not belong to the face f_j, say vertices v and w in 5.5(b). Then the graph is no longer planar as assumed from the beginning. Now consider the case where the path connect two vertices, say u and w shared by the face f_j and the sets V_{ab} and V_{bc} respectively. Then faces f_i and f_j no longer share the vertex b and the sets V_{ab}, V_{bc} and V_{ca} do not define a partition of $V(G)$. □

Thus we have proved the following result.

Corollary 5.1. Algorithm 3 in Chapter 4 computes a set decomposition of a biconnected graph, if one exists.

5.5 Chapter Summary

The new DR-planner tree-decomposition algorithm described is correct. First, we show that transforming the input graph into the merged graph preserves hinges. Then, we proved the resulting merged graph is planar. Finally, we proved the main result, a merged graph is tree decomposable if and only if there is a planar embedding with two faces which share three or more vertices.

CHAPTER 6

Complexity

> Success consists of going from failure to failure without loss of enthusiasm.
>
> *(Winston Churchill)*

In Chapter 4 we presented a new direct decomposition algorithm for geometric constraint graphs related to well-constrained geometric constraint solving problems. The algorithm was described in rather abstract steps. In this chapter some of those steps are further developed to allow reasoning about their complexity. Given an input graph $G = (V, E)$ we will use the number of vertices $|V|$ as the unit to measure complexity. However, according to Laman Theorem 3.1, the relation between the number of edges and vertices in well-constrained graphs is linear and thus, from an asymptotic standpoint, the number of vertices and the number of edges both are equivalent measures.

We prove first that given a well-constrained biconnected input graph $G = (V, E)$, the worst time complexity of our basic tree decomposition step DR-planner algorithm is $O(|V|^2)$. Then we analyze the running time for the whole recursive DR-planner and prove that the worst time complexity is $O(|V|^2)$. We apply the usual unit-cost operations, unbounded memory random access machine computational model which have unit cost for read and write access to all of its memory cells, [104].

Most of the work of this chapter has been previously published in reference [71].

6.1 Graph-connectivity

The first step of the new Algorithm 2 given in Chapter 4 is to determine the connectivity of the input graph, that is, whether it is either 0-connected, 1-connected or a biconnected graphs. Then the graph is decomposed using a specific algorithm according to its connectivity. To classify an input graph G into some of the classes we proceed as shown in Algorithm 2 of Chapter 4.

To determine the connectivity of a graph G we compute the associated forest of depth first trees. If the forest contains several trees, then G is a 0-connected graph. If the forest contains

Algorithm 4 Decomposition of 0-connected graphs.

procedure 0-con-decomp
INPUT
    0-connected constraint graph $G = (V, E)$ with $|V| \geq 3$
OUTPUT
    A set decomposition of G
Let T be a connected component of G
if $|V(T)| > 1$ **then**
    Let $v_1, v_2 \in V(T)$
    Let $v_3 \in V - V(T)$
    $G_1 = V(T)$; $G_2 = (V - V(T)) \cup \{v_1\}$; $G_3 = \{v_2, v_3\}$
else
    /* T is a component with one vertex */
    Let $v_1, v_2 \in V - V(T)$
    Let $v_3 \in V(T)$
    $G_1 = V - V(T)$; $G_2 = \{v_1, v_3\}$; $G_3 = \{v_2, v_3\}$
endif
return $\{G_1, G_2, G_3\}$

just one tree, then G is either 1-connected or biconnected. Finally 1-connected graphs are distinguished from biconnected graphs by searching for an articulation vertex in the tree, [51]. If found, G is 1-connected otherwise G is biconnected.

We start by introducing Lemma 6.1 concerning the run time complexity of the classification procedure described above.

Lemma 6.1. *Let $G = (V, E)$ a graph. To classify G as a 0-connected, 1-connected or biconnected can be done in $O(|V|)$ worst case asymptotic time.*

Proof. The forest of G can be computed in $O(|V|)$ by applying a DFS traversal of G. An articulation vertex, if any, can be computed in $O(|V|)$ using the algorithm described in [51]. □

In what follows we study the complexity of the algorithm for each graph class.

6.2 Complexity of Decomposition of 0-connected Graphs

To find a tree-decomposition of a 0-connected graph, all what we have to do is to properly group connected components in the given graph into three subgraphs pairwise sharing one vertex. See Algorithm 4. The complexity of this algorithm is stated in Lemma 6.2.

Lemma 6.2. *Let $G = (V, E)$ be a 0-connected graph with $|V| \geq 3$. The worst case asymptotic time complexity of computing the decomposition of G is $O(|V|)$.*

Proof. Assume that G is represented using an adjacency list. In this case the decomposition algorithm requires to compute the set $\mathcal{K} = \{K_1, \ldots, K_m\}$ of connected components of G and then select a proper subset of $\mathcal{K}$ with at least two vertices. This can be done in linear time and thus its complexity is $O(|V|)$. □

Algorithm 5 Decomposition of 1-connected graphs.

procedure 1-con-decomp
INPUT
A 1-connected constraint graph $G = (V, E)$ with $|V| \geq 3$
OUTPUT
A set decomposition of G
Select one articulation vertex $v_1 \in V$
Let $G_1 = G - v_1$
Let $K_1, K_2 \subset G_1$ two arbitrary connected components
Let $v_2 \in V(K_1)$ and $v_3 \in V(K_2)$
$G_1 = V_1 \cup \{v_1\}$, $G_2 = V_2 \cup \{v_1\}$, $G_3 = \{v_2, v_3\}$
return $\{G_1, G_2, G_3\}$

6.3 Complexity of Decomposition of 1-connected Graphs

A procedure to compute the tree-decomposition of a 1-connected graph is given in Algorithm 5.

As shown by Lemma 6.3, the worst case running time is $O(|V|)$.

Lemma 6.3. *Let $G = (V, E)$ be a 1-connected graph. The worst case asymptotic time cost of computing the decomposition of G is $O(|V|)$.*

Proof. Assume G is represented using an adjacency list. When G is 1-connected, in $V(G)$ there is at least an articulation vertex. Following Hopcroft and Tarjan, [51], the asymptotic cost of computing one articulation vertex is $O(|V|)$. Assume that $v_1 \in V$ is an articulation vertex, then removing v_1 breaks the graph G into several subgraphs $\mathcal{K} = \{K_1, \ldots, K_m\}$. Choose some subgraph, say K_1, and choose an arbitrary vertex $v_2 \in K_1$ and some other vertex $v_3 \in K_2 \cup \cdots \cup K_m$. Then v_1, v_2 and v_3 are hinges of G. Then the overall worst case asymptotic cost of this decomposition is $O(|V|)$. □

6.4 Complexity of Decomposition by Deleting degree 2 vertices

Subgraphs in biconnected graphs with degree 2 vertices can be solved in a direct specific way applying the Todd method described in Chapter 4.4.1.

The cost of deleting the degree 2 vertices of a graph G is asserted in Lemma 6.4.

Lemma 6.4. *Let $G = (V, E)$ be a graph. The cost of deleting vertices with degree 2 from a graph is $O(|V|)$.*

Proof. See Todd, [124]. □

6.5 Complexity of Decomposition of Biconnected Graphs

When the graph at hand is biconnected, we compute a basic tree-decomposition with the Algorithm 3 introduced in Chapter 4. We first figure out the associated worst running time for each step in the algorithm.

6.5.1 The Set of Fundamental Circuits

Given a biconnected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. The problem of computing the set of fundamental circuits amounts to computing first an spanning tree of G then identifying the associated fundamental circuits in it. Lemma 6.5 refers to the complexity of computing the set of fundamental circuits.

Lemma 6.5. *Let $G = (V, E)$ be a biconnected graph. The cost of compute a spanning tree T to G and then compute the set of associated fundamental circuits to G is $O(|V|)$.*

Proof. Both the spanning tree T and the set of fundamental circuits are computed by applying a depth-first search algorithm. The spanning tree T is computed in a worst case time given by $O(|V| + |E|)$, [101]. Identifying the fundamental circuits as cycles in T amounts to, at most, visiting all edges in E, therefore, the running time is $O(|E|)$. Since for well-constrained and under-constrained graphs, the Laman relation can be written as $|E| \leq 2|V| - 3$, [81], we conclude that the worst case running time for computing the spanning tree and fundamental circuits is $O(|V|)$. □

6.5.2 The Set of Bridges Implementation

Algorithm 6 used to compute the set of bridges is a depth-first search slightly modified to account for halting recursive calls when reaching bridge attachments. In the procedure given in Algorithm 6, e denotes an edge in $E(G)$ bounded by vertices $e.v, e.w \in V(G)$. Usual notation for LIFO and FIFO data structures has been used.

Lemma 6.6 gives an upper bound for the cost of computing the bridges.

Lemma 6.6. *Let $G = (V, E)$ be a biconnected well-constrained graph and let C be a fundamental circuit of G. The cost of computing the bridges induced by C in G is $O(|V|)$.*

Proof. Algorithm 6 has two steps. In the first one the set of singular bridges, that is, those bridges which include just one edge incident on two vertices in $V(C)$ is computed. Trivially, the running time is $O(|V(C)|)$. The second step of the algorithm is a depth-first search approach modified to account for halting the backtracking when a vertex in $V(C)$ is reached. Therefore the overall cost worst running time is $O(|V|)$. □

6.5.3 The Collapsed Graph

The goal of this stage is to compute the collapsed graph. For each bridge we check whether it is singular or not and we convert it to a star graph. If the bridge is singular we add a new vertex and two edges otherwise we consider only the attachments of the bridge and add a new vertex and the connecting edges. Algorithm 7 shows the procedure.

Lemma 6.7 shows the worst running time for the Algorithm 7.

Lemma 6.7. *Let $G = (V, E)$ be a biconnected well-constrained graph and let C be a fundamental circuit of G. Let $B = \{B_1, \ldots, B_k\}$ be the set of bridges induced by C. The cost of computing the collapsed graph G' of G is $O(|V|)$.*

Algorithm 6 Computing the set of bridges.

```
INPUT:
    G = (V, E): Biconnected constraint graph with |V| > 3
    C: Fundamental circuit of G
OUTPUT:
    B: The set of bridges induced by C in G
B = Set of singular bridges
Remove from E(G) the set of edges in B
S = emptystack
Label vertices in V(G) as not visited
for each vertex v in V(G) − V(C) not visited do
    Label v as visited
    B_i = emptyset
    for all edges e in E(G) − E(C) adjacent to v do
        S.push(e)
    endfor
    while not S.empty() do
        e = S.pop()
        B_i = B_i ∪ ({e.v, e.w}, e)
        for all edges a in E(G) − E(C) adjacent to e.w and
            a.w not visited and a not in E(C) do
                S.push(a)
        endfor
    endwhile
    B = B ∪ B_i
endfor
return B
```

Proof. Collapsed bridges B' are computed by collapsing each bridge in B. Clearly the worst running time needed to collapse the bridges in B is $O(|B|)$. The number of attachments in a singular bridge is two and for a non-singular bridge the number is two plus, at least, one internal bridge attachment. That is, for each bridge $B_i \in B$, we have $|V(B_i)| \geq 2$. Hence the total number of bridges is $|B| \leq |V(G)/2|$. Computing the collapsed graph entails visiting once each vertex in $V(G) - V(C)$. Since $|B| \leq |V(G)/2|$ and since the total number of visited vertices is $|V(G) - V(C)| < |V(G)|$, the total running time needed to collapse bridges is $O(|V|)$. □

6.5.4 The Merged Graph

To compute the merged graph we need to identify in the collapsed graph clusters of star bridges which interlace and actually merge them into one single graph.

A naive quadratic approach to identify clusters of star bridges which interlace would consist on visiting each star and checking for interlacements with other stars. The approach we have adopted is based on the scan-line algorithm, [30] described by Miller and Ramachandran, [92],

Algorithm 7 Algorithm to compute the collapsed graph.

INPUT
$G = (V, E)$: Biconnected constraint graph with $|V| > 3$,
B the set of bridges induced by C
OUTPUT
Collapsed Graph G' of G

$C = \emptyset$
foreach $B_i \in B$ **do**
  if $singular(B_i)$ **then**
    $V(B_i) = at(B_i) \cup \{x\}$
    $E(B_i) = ((x, First(at(B_i))) \cup (x, Last(at(B_i))))$
  else
    $V(B_i) = at(B_i) \cup \{x\}$
    $E(B_i) = \emptyset$
    foreach $at_i \in at(B_i)$ **do**
      $E(B_i) = (x, at_i) \cup E(B_i)$
    endfor
  $C = C \cup (V(B_i), E(B_i))$
endfor
return C

to find the merging of a set of star graphs. The approach performs a single traversal of the vertices in the fundamental circuit under consideration resulting in a linear procedure. Miller and Ramachandran reported in [92] a scan-line linear algorithm to find the merging of a set of star graphs. As described, we understand that as the algorithm scans the set of sorted edges, stars stored in the stack must be sorted from bottom to top according to decreasing indexes of the last attachment of each star graph. This ordering defines an invariant the algorithm must preserve. As far as each vertex in the circuit C is an attachment for at most one star edge, the order defined on the set of vertices and the one defined on the set of edges are coincident and just scanning edges automatically preserves the stack invariant. However, when two or more edges belonging to different stars in the input graph share as attachment a vertex in the circuit C, the stack may store stars which do not interlace with the star the edge at hand belongs to. In this scenario, the stack invariant no longer holds if the star is directly pushed on the stack.

Example 6.1. To illustrate the situation consider the graph example in Figure 6.1. The set of sorted edges is $\{e_0, e_4, e_2, e_6, e_5, e_3, e_1, e_7\}$. After visiting edges e_0 and e_4, the stack stores from bottom to top star bridges B_0 and B_2. When considering edge e_2, we found that star bridge B_1 does not interlace with any star in the stack. If star bridge B_1 is pushed onto the stack, the stack invariant no longer holds and the algorithm fails.

To deal with problems where star bridges attachments are common to more than one star bridge in the input graph, we have generalized the approach reported in [92]. Our method is outlined in Algorithm 8-Algorithm 10.

Algorithm 8 Main algorithm to compute the the merging of a collapsed graph.

INPUT:
  CG: Collapsed graph as a set of star bridges
OUTPUT:
  MG: The merged graph as a set of merged bridges
Merge all star bridges in graph CG with the same span
MG = emptyset
S = emptystack
L=$sortEdges(E(CG))$
e=$L.front$
B=star bridge to which e belongs to
$S.push(B)$
for each vertex edge e in L **do**
  if e is the last edge of a bridge **then**
    $MG = MG \cup S.pop()$
  elif e is an internal edge of a bridge **then**
    $DispatchInternalEdge(e)$
  elif e is the first edge of a bridge **then**
    $DispatchFirstEdge(e)$
  endif
endfor
Remove multiple edges in MG
return MG

The algorithm uses the following data structures: two bucket-sorted tables to store respectively bridges and edges, and a stack to dynamically manage the set of bridges currently under consideration. Additionally, we use a stack of bridges for housekeeping. So far no attempt has been made to minimize the amount of storage used by the algorithm. Function *isTheFirstBridge(bridge,B)* in the Algorithm 10 returns true if and only if the last edge of the popped bridge, *bridge*, is no smaller than the last edge of the current bridge, B. See [92].

The main Algorithm 8, first merges bridges with same span. Next, edges are sorted in increasing order of their attachment in $V(C)$. Edges with the same attachment are sorted in increasing order of the last edge of the bridge they belong to with ties broken by ordering in decreasing order of the first edge of bridges they belong to, [92]. Since no two bridges have the same span, ties are always broken. Then the bridge to which the first edge in the sorted edges belongs to is pushed on the stack. The remaining edges are sequentially visited and dispatched according to whether the edge is the last edge, an internal edge or the first edge in the bridge they belong to.

The algorithm leverages from merging at the beginning bridges with the same span in three ways. First bridges leading to faces in the final planar embedding whose attachments do not contribute to the set of hinges are removed.

Example 6.2. Consider a circuit C with bridges B_i and B_j such that $sp(B_i) = sp(B_j)$ and

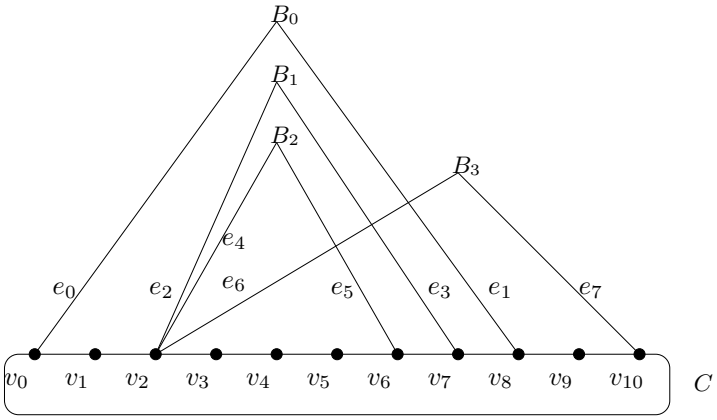


Figure 6.1: Graph Example.

$B_j \subset B_i$ as depicted in Figure 6.2(a). Clearly, vertices common to $at(B_i)$ and $V(C)$ are just the first and last vertices in $at(B_i)$ which are coincident respectively with the first and last vertices in $at(B_j)$. Therefore the contribution of bridge B_i to the set of hinges is subsumed by B_j and there is no need for further considering B_i.

Then it is easy to see that if two bridges, say B_i and B_j, share three attachments and at least one of them has more than three vertices, the first part of the interlacing definition given in Chapter 2.8 holds. Thus, the second part of the definition concerning bridges which share three attachments is introduced just to consider the case where bridges B_i and B_j have both exactly three attachments common to both bridges.

See Figure 6.2(b). From the point of view of how bridges B_i and B_j contribute to the existence of hinges, clearly we only need to consider one of them. Thus merging them does not result in the loose of any hinge. Besides, the merging of these bridges frees our merging algorithm from the need of checking for the second part of the interlacing definition.

Finally, if $sp(B_i) = sp(B_j)$ and B_i or B_j or both have four or more attachments, then either $at(B_i) = at(B_j)$ and one of the bridges can be removed or B_i and B_j interlace and can trivially be merged.

Now the complexity for the computation of the merged graph is stated by Lemma 6.8.

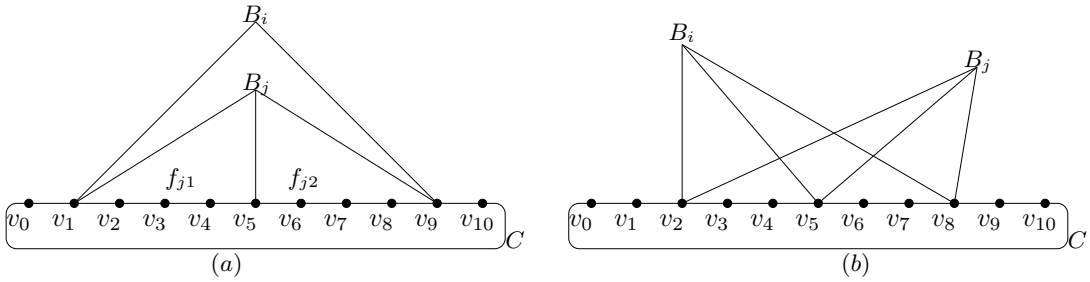


Figure 6.2: (a) Bridge B_j subsumes the contribution to hinges of bridge B_i. (b) Bridges with exactly three attachments which are common to both bridges.

Lemma 6.8. *Let $G = (V, E)$ be a collapsed graph with a set of n bridges $B = \{B_1, \ldots, B_n\}$. The computation of the merged graph is $O(|V|)$.*

Proof. The set of merged bridges $\{B''_1, \ldots, B''_n\}$ is computed by applying to the set of collapsed bridges B' the sequential scan-line Algorithm 8. This algorithm merges a set of star subgraphs where interlacing occurs and is implemented in linear time with respect to the total number of stars attachments. In our case, the graph is derived from a well-constrained Laman graph therefore the number of attachments is at most $|V(G)|$ and the worst running time is $O(|V|)$. □

6.5.5 The Merged Graph Embedding Implementation

The next stage is to compute a planar embedding for the planar merged graph G'' induced in the input graph G by a fundamental circuit $C = \langle v_1, v_2, \ldots, v_n \rangle$.

Many general algorithms to compute embedding of planar graphs have been published. See, for example, [95, 96]. However we are interested in finding a specific embedding. According to the Jordan Curve Theorem, [24], embedding the circuit C in the plane results in a planar embedding with two faces, one bounded face, f_b and one unbounded face, f_u. We want our embedding to preserve the bounded face f_b.

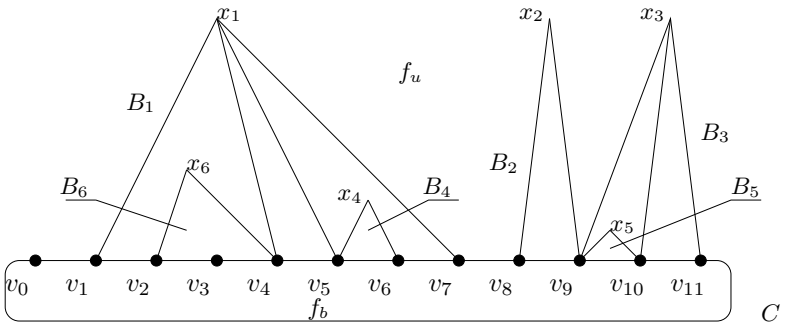


Figure 6.3: Labeling of circuit and bridges attachments.

We assume that circuit vertices $V(C)$ are circularly labeled with increasing label values and that bridges attachments are accordingly sorted. Moreover, we assume if the circuit under consideration is $C =< v_1, v_2, \ldots, v_n >$, the attachments of each bridge $at(B_i) = \{v_{i1}, v_{i2}, \ldots, v_{im}\}$ fulfill $v_1 \leq v_{i1}$ and $v_{im} \leq v_n$. See Figure 6.3. Notice that this assumption entails, at most, a coherent relabeling of vertices in the circuit and bridges. Our algorithm first sorts edges in the merged graph in increasing order of their attachments in circuit C. Notice that given a vertex $v \in V(C)$, the number of edges incident on v which are internal to some bridge is at most one. Otherwise, the merged graph would have at least two interlacing star bridges. See Algorithm 11, where $L = L + outList + xList + inList$ must be understood as a list append.

Then, for each face in the planar embedding, the scan-line sequential Algorithm 12 extracts the set of vertices that the face shares with the circuit vertices *V(C)*.

The algorithm is based on two observations. Fist notice that any edge incident on a vertex $v \in V(C)$ is either the first edge of a bridge, an internal edge or, the last edge of a bridge. If the edge is the first one a new face in the embedding is found. If the edge is internal to a

bridge then a face has been completed and a new one has been found. Finally, if the edge is the final edge of a bridge, a face has been completed.

Now note that, given a face of the planar embedding, we want to identify the set of vertices common to the face at hand and the face bounded by the circuit C. Consider two bridges, say B_i and B_j, with spans $sp(B_i)$ and $sp(B_j)$ such that $sp(B_j) \subseteq sp(B_i)$. Clearly, faces generated by bridge B_i cannot include vertices in $sp(B_j)$. For example, in the merged graph shown in Figure 6.3, faces derived from bridge B_1 can not include vertex v_3. To formalize this idea, we define the concept of *covered vertex* as follows. We say that a vertex $v \in V(C)$ is covered by the bridge B in the merged graph if $v \in sp(B)$. To capture the covering property, the algorithm associates with each vertex $v \in V(C)$ a label which initially identifies the vertex v and that takes value $'x'$ whenever vertex v is covered by a bridge.

Example 6.3. As a general example, consider the set of bridges $B_1, \ldots, B_6$ in Figure 6.3. The algorithm described would report the set of faces $f_1 = \{v_2, v_3, v_4\}$, $f_2 = \{v_1, v_2, x, v_4\}$, $f_3 = \{v_4, v_5\}$, $f_4 = \{v_5, v_6\}$, $f_5 = \{v_5, v_6, v_7\}$, $f_6 = \{v_8, v_9\}$, $f_7 = \{v_9, v_{10}\}$, $f_8 = \{v_9, v_{10}\}$ and $f_9 = \{v_{10}, v_{11}\}$. The unbounded face if $f_u = \{v_1, x, x, x, x, x, v_7, v_8, v_9, x, v_{11}\}$.

Finally Lemma 6.9 gives the cost of computing the embedding of the merged graph.

Lemma 6.9. *Let $G = (V, E)$ be a merged graph. The cost of embedding the merged graph is $O(|V|)$.*

Proof. Computing the planar embedding includes two steps. First the set of edges is sorted by Algorithm 11. Each vertex in the circuit $V(C)$ is visited once and the algorithm can be implemented in linear time with a bucket sort. Then each face in the planar embedding is extracted by the Algorithm 12 as the set of vertices the face shares with $V(C)$. Edges in the merged graph which are either the first or the last edge in a bridge are visited once while each edge internal to a bridge is visited twice. Since the merged graph is planar, the Euler relation $|V| + f - |E| = 2$, where f is the number of faces, holds [28]. The number of faces in f is fixed once $|V|$ and $|E|$ are fixed. Therefore the worst running time for computing the planar embedding is $O(|V|)$. □

6.5.6 Computing the Hinges

Algorithm 13 shows the methodology to find the hinges for the given graph G by searching in the planar embedding of the merged graph G'''.

The complexity of this algorithm is stated in Lemma 6.10.

Lemma 6.10. *Let $G = (V, E)$ be a biconnected graph, let C be a fundamental circuit of G and let G'' be the merged graph of G induced by C. The cost of find hinges in the plane merged graph D of G'' is $O(|V|)$.*

Proof. Given a plane merged graph D associated to one fundamental circuit C of G, the cost of exploring the faces of the embedding D to find the hinges is to perform a search for each face to find a face $F \in D$ with 3 circuit vertices belonging to $F \cap F_b$ as shown in Algorithm 13. That cost is linear to the number of faces. And according to the Euler relation $|V| + f - |E| = 2$, the cost of exploring the faces is linear to the number of vertices of G. □

6.5.7 The Basic Tree-decomposition Algorithm

Lemma 6.11. *Given a well-constrained biconnected graph $G = (V, E)$, the basic tree-decomposition step algorithm is $O(|V|^2)$.*

Proof. The basic tree-decomposition step Algorithm 3 in Chapter 4, loops over the set of fundamental circuits in the worst case. Then, for each circuit, the algorithm loops over the set of faces in the planar embedding. The number of vertices bounding a face is $|V|$. Therefore, the worst case running time is $O(|V|^2)$. □

6.6 The Tree-decomposition DR-Planner Algorithm Running Time

Our DR-planner computes the decomposition plan by recursively applying basic tree-decomposition steps. We have the following result.

Theorem 6.1. *Given a tree-decomposable biconnected graph, $G = (V, E)$, the DR-planner based on tree-decomposition Algorithm 2 in Chapter 4 computes a tree-decomposition in $O(|V|^2)$.*

Proof. Clearly, if the constraint graph $G = (V, E)$ is such that $|V| = 3$, the total amount of work is constant, say $T(3)$.

Consider now biconnected graphs with $|V| > 3$. At each decomposition step our DR-planner divides the constraint graph $G = (V, E)$ into three subgraphs, $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ and $G_3 = (V_3, E_3)$. Under the assumption that the biconnected constraint graph is well-constrained, the number of vertices in each subgraph, say $|V_i|$, is roughly $|V|/3$. We have shown that the amount of work done outside the recursive calls is $O(|V|^2)$. Therefore the total amount of work done by our recursive DR-planner can be expressed as a function of the cardinality of the set of vertices, $|V|$, as

$$T(|V|) = 3T(\frac{|V|}{3}) + O(|V|^2)$$

Assume that the number of decomposition steps is m. Thus, we can expand out the recurrence relation yielding

$$T(|V|) = |V|^2 + \frac{|V|^2}{3} + \frac{|V|^2}{3^2} + \frac{|V|^2}{3^3} + \cdots + \frac{|V|^2}{3^{m-1}} + T(3)$$

Where $T(3)$ is the work needed to decompose the basic constraint graph $G = (V, E)$ with $|V| = 3$. Then

$$T(|V|) = |V|^2 + \sum_{i=0}^{m-1} \frac{1}{3^i} + T(3)$$

Therefore, in terms of Big O notation, we have $T(|V|) = O(|V|^2)$. Since this recurrence relation follows the pattern $T(n) = aT(n/b) + n^c$, where $a = 3$, $b = 3$ and $c = 2$. Hence we have that $c > log_b(a) + \epsilon$, with $\epsilon > 0$, and according to the generalization of the the Master Theorem [23] proposed by Akra and Bazzi in [3], it is straightforward to confirm that the work done by our recursive $DR - planner$ is $T(|V|) = O(|V|^2)$. □

6.7 Chapter Summary

Our approach based on the tree-decomposition of a constraint graph G has a worse case $O(|V|^2)$ computation cost. This result assures that the algorithm is competitive with the existing decomposition algorithms in these area.

Algorithm 9 Dispatching an edge internal to a star bridge.

```
INPUT
    e: edge under consideration
OUTPUT
    Updates the bucket-sorted table of edges
    Updates the bucket-sorted table of current merged bridges
B=star bridge to which e belongs to
tomerge = emptyset
Saux=emptystack
bridge = S.pop()
while bridge ≠ B do
    if not interlace(bridge, B) then
        Saux.push(bridge)
    else
        tomerge = tomerge ∪ {bridge}
    endif
    bridge = S.pop()
endwhile
if tomerge ≠ ∅ then
    mergedbridge=mergeBridges(tomerge)
    S.push(mergedbridge)
    for each bridge in tomerge do
        Remove bridge from bucket-sorted bridges table
    endfor
    Include mergedbridge in bucket-sorted bridges table
    Update edges in the bucket-sorted according to edges in mergedbridge
else
    S.push(B)
endif
while not Saux.empty() do
    S.push(Saux.pop())
endwhile
```

Algorithm 10 Dispatching the first edge of a star bridge.

```
INPUT
    e: edge under consideration
OUTPUT
    Updates the bucket-sorted table of edges
    Updates the bucket-sorted table of current merged bridges
B=star bridge to which e belongs to
tomerge = emptyset
Saux = emptystack
bridge = S.pop()
while not isTheFirstBridge(B, bridge) do
    if not interlace(bridge, B) then
        Saux.push(bridge)
    else
        tomerge = tomerge ∪ {bridge}
    endif
    bridge = S.pop()
endwhile
if tomerge ≠ ∅ then
    mergedbridge=mergeBridges(tomerge)
    S.push(mergedbridge)
    for each bridge in tomerge do
        Remove bridge from bucket-sorted bridges table
    endfor
    Include mergedbridge in bucket-sorted bridges table
    Update edges in the bucket-sorted according to edges in mergedbridge
else
    S.push(B)
endif
while not Saux.empty() do
    S.push(Saux.pop())
endwhile
```

Algorithm 11 Sorting the set of edges of the merged graph implementation.

```
INPUT
    E: Set of edges of the merged graph
    V(C): Set of vertices of circuit C
OUTPUT
    L: List of sorted edges
L = emptylist
for each vertex v in V(C) do
    inList,outList=emptylist
    for each edge e incident with v do
        if e is the first edge of a star bridge then
            Insert e in inList according to increasing values of
            the last edge of star bridges
        elif e is the last edge of a star bridge then
            Insert e in outList according to decresaing values of
            the first edge of star bridges
        elif e is an internal edge of a star bridge then
            xlist=e
        endif
    endfor
    L=L + outList + xList + inList
endfor
return L
```

Algorithm 12 Computing the embedding of the merged graph while preserving the face bounded by the circuit C.

INPUT
    L: List of sorted edges in the merged graph
OUTPUT
    F: Planar embedding as a list of faces
F = emptylist
S = emptystack
Label each vertex v in *V(C)* as not covered
for each vertex edge e in L **do**
    if e is the first edge of a bridge **then**
        S.push(e)
    elif e is the last edge of a bridge **then**
        e_i=*S*.pop(e)
        Define face f as the set of labels associated with vertices
            $\{v \in V(C) | e_i.c \leq v \leq e.c\}$
        $F = F \cup \{f\}$
        Label as covered the vertices in the set $\{v \in V(C) | e_i.c < v < e.c\}$
        if e_i is an internal edge of a bridge **then**
            Label e_i as covered
        endif
    elif e is an internal edge of a bridge **then**
        $e_i = S.pop(e)$
        Define face f as the set of labels associated with vertices
            $\{v \in V(C) | e_i.c \leq v \leq e.c\}$
        $F = F \cup \{f\}$
        Label as covered the vertices in the set $\{v \in V(C) | e_i.c < v < e.c\}$
        if e_i is an internal edge of a bridge **then**
            Label e_i as covered
        endif
        S.push(e)
    endif
endfor
Define face f_u as the set of labels associated with $V(C)$
$F = F \cup \{f_u\}$
return F

Algorithm 13 Algorithm to find the hinges.

procedure findHinges
INPUT
A set of Faces $F = \{f_1, \ldots, f_n\} \cup f_b$, where f_b is the face bounded by the circuit
OUTPUT
$\{v_1, v_2, v_3\}$: a set of hinges, if one exists
$hinges = \emptyset$
for f_i **in** F **do**
Compute $\nu = V(f_i) \cap V(f_b)$
if $|\nu| \geq 3$ **then**
hinges= any $\{v_1, v_2, v_3\} \subseteq \nu$
return *hinges*
return $\emptyset$

CHAPTER 7

Conclusions and Future Research Directions

I've been a lucky man in life,
nothing was easy for me.

(Sigmund Freud)

Conclusions

In this thesis we have developed a new strategy to solve geometric constraint problems in the Euclidean space based on a decomposition-recombination approach known as tree decomposition. Tree decomposition is a graph-based technique that recursively splits the constraint graph into three subgraphs such that pairwise share one vertex. These shared vertices are called hinges. The strategy is not general but allows to solve a sizable amount of problems belonging to the quadratically solvable problems also known as ruler-and-compass solvable problems, which are usually found in practice in a number of important scenarios, for example in CAD/CAM and related fields.

In what follows we summarize our contributions and list a set of open challenges we plan to pursue in the near future.

Research Contributions

We have developed a new graph-based DR-planner that solves geometric constraint problems in the Euclidean space by tree-decomposing the constraint graph which abstracts the geometric problem. The DR-planner developed systematically searches for hinges in fundamental circuits associated to a spanning tree of the constraint graph and consists of two main steps. First the graph is transformed into a simpler, planar graph. Then the transformed graph is embedded in the plane where hinges are identified as a set of three vertices shared by the embedding of the face bounded by a fundamental circuit and any other face in the embedding.

A number of general and efficient algorithms to embed planar graphs in the plane have been published. However, for the sake of efficiency, we have developed a specific algorithm tailored to our needs. The implemented algorithm takes advantage of the natural hierarchy induced by the

set of bridges in the planar graph. Furthermore, it guarantees that the resulting embedding always includes the face bounded by the fundamental circuit considered. The algorithm is conceptually simple and easy to implement.

The DR-planner is correct in the following sense. First the graph transformation preserves the hinges. Second, hinges, if present, always belong to the common boundary of two faces in a planar embedding of the transformed constraint graph. The fact that one of the faces involved in the search is always the bounded by the considered fundamental circuit in the planar embedding facilitates the search.

Because the approach is based in simple concepts, implementation requires simple data structures. Notice that although the approach computes a graph tree decomposition, no need for specific geometric construction steps are needed. The approach identifies sets of three vertices that conceptually decompose the graph at hand thus all what it requires is that the input graph be structurally either well-constrained or under-constrained. Therefore the algorithm yields a tree-decomposition even in the case where there are constraints in the problem which fulfill geometric theorems, for example, three lines pairwise constrained by angles.

When in a tree-decomposition step the input graph is over-constrained including either well-constrained or under-constrained subgraphs the algorithm correctly decomposes the input graph and the over-constrained part goes within one of the resulting clusters. However if the input graph does not include either well-constrained or under-constrained subgraphs no planar embeddings exists which includes two faces with three or more common vertices. Therefore no hinges exist and the algorithm issues the corresponding warning.

The worst case overall performance of the DR-solver built on top of our DR-planner is quadratic. This does not improve over other tree-decomposition DR-solvers previously reported in the literature. However, we have applied our implementation to tree-decomposable constraint graphs with up to one thousand nodes and the measured running time is clearly smaller than quadratic.

Summarizing, the proposed approach is a viable alternative to already developed tree-decomposition based DR-planners which operate by identifying specific subgraphs in the constraint graph. The systematic nature allows a clean and easy implementation.

Future Research Directions

We have in mind two different avenues to explore in the near future: to improve the DR-planner efficiency and to widen the solver domain, that is to enlarge the set of problems the DR-planner is able to solve.

We believe that there is room left to improve the practical performance of our DR-planner. First we have observed that under given circumstances, intermediate data resulting when considering a fundamental circuit that does not include hinges do not need to be recomputed when considering other circuits. Identifying a relationship between these intermediate results would help in avoiding duplicate calculations. To further improve the performance, we plan to develop a parallel version of our approach. Distributing fundamental circuits is an *a priori* plausible strategy.

So far the geometric elements a geometric constraint problem can include are points, straight

lines, circles and arcs of circle with constant radii. Extending our approach to solve problems including other geometric elements like parabolas, ellipses or splines would be a great accomplishment.

Bibliography

[1] Ait-Aoudia, S., and Foufou, S. A 2D geometric constraint solver using a graph reduction method. *Advances in Engineering Software 41*, 10-11 (Oct. 2010), 1187–1194. http://dx.doi.org/10.1016/j.advengsoft.2010.07.008.

[2] Ait-aoudia, S., Jegou, R., and Michelucci, D. Reduction of constraint systems. In *3rd International Conference on Computational Graphics and Visualization Techniques* (1993), Compugraphics'93, Alvor, Algarve : Portugal, pp. 83–92.

[3] Akra, M., and Bazzi, L. On the solution of linear recurrence equations. *Computational Optimization and Applications 10*, 2 (May 1998), 195–210.

[4] Aldefeld, B. Variation of geometrics based on a geometric-reasoning method. *Computer-Aided Design 20*, 3 (Apr. 1988), 117–126. http://dx.doi.org/10.1016/0010-4485(88)90019-X.

[5] Allgower, E. L., and Georg, K. Continuation and path following. *Acta Numerica 2* (Jan. 1993), 1–64. http://dx.doi.org/10.1017/S0962492900002336.

[6] Bettig, B., and Hoffmann, C. M. Geometric constraint solving in parametric computer-aided design. *Journal of Computing and Information Science in Engineering 11*, 2 (2011). http://dx.doi.org/10.1115/1.3593408.

[7] Bhansali, S., Kramer, G. A., and Hoar, T. J. A principled approach towards symbolic geometric constraint satisfaction. *Journal of Artificial Intelligence Research 4* (1996), 419–443. http://dx.doi.org/10.1613/jair.292.

[8] Bona, M., Sitharam, M., and Vince, A. Enumeration of viral capsid assembly pathways: Tree orbits under permutation group action. *Bulletin of Mathematical Biology 73* (2011), 726–753. http://dx.doi.org/10.1007/s11538-010-9606-4.

[9] Bondy, J. A. *Graph Theory With Applications*. Elsevier Science Ltd., Oxford, UK, UK, 1976. ISBN: 0444194517.

[10] Borning, A. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems 3*, 4 (Oct. 1981), 353–387. http://dx.doi.org/10.1145/357146.357147.

[11] Borning, A., Anderson, R., and Freeman-Benson, B. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 1996), UIST '96, ACM, pp. 129–136. http://dx.doi.org/10.1145/237091.237110.

[12] Bouma, W., Fudos, I., Hoffmann, C., Cai, J., and Paige, R. Geometric constraint solver. *Computer-Aided Design 27*, 6 (1995), 487 – 501. http://dx.doi.org/10.1016/0010-4485(94)00013-4.

[13] Brüderlin, B. Constructing three-dimensional geometric objects defined by constraints. In *Proceedings of the 1986 workshop on Interactive 3D graphics* (New York, NY, USA, 1987), I3D '86, ACM, pp. 111–129. http://dx.doi.org/10.1145/319120.319130.

[14] Brüderlin, B. *Rule-based geometric modelling.* PhD thesis, Informatik-Dissertationen ETH Zürich, 1988.

[15] Brüderlin, B. Symbolic computer geometry for computer aided geometric design. In *Advances in Design and Manufacturing Systems* (Tempe, AZ, Jan. 8-12 1990), Proceedings NSF Conference.

[16] Brüderlin, B. Using geometric rewrite rules for solving geometric problems symbolically. *Theoretical Computer Science 116*, 2 (Aug. 1993), 291–303. http://dx.doi.org/10.1016/0304-3975(93)90324-M.

[17] Buchanan, S. A., and de Pennington, A. Constraint definition system: a computer-algebra based approach to solving geometric-constraint problems. *Computer-Aided Design 25*, 12 (1993), 741 – 750. http://dx.doi.org/10.1016/0010-4485(93)90101-S.

[18] Buchberger, B. *Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory.* Reidel Publishing Company, Dodrecht - Boston - Lancaster, 1985.

[19] Chiang, C.-S., and Joan-Arinyo, R. Revisiting variable radius circles in constructive geometric constraint solving. *Computer-Aided Geometric Design 21*, 4 (2004), 371 – 399. http://dx.doi.org/10.1016/j.cagd.2004.01.003.

[20] Chou, S. An introduction to Wu's method for mechanical theorem proving in geometry. *Journal of Automated Reasoning 4*, 3 (1988), 237–267.

[21] Chou, S.-C., and Gao, X.-S. Automated generation of readable proofs with geometric invariants i. multiple and shortest proof generation. *Journal of Automated Reasoning 17*, 3 (1996), 325–347.

[22] Condoor, S. S. *Mechanical Design Modeling Using Pro/Engineer*, 1st ed. McGraw-Hill Higher Education, 2001. ISBN: 0072443146.

[23] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company, 1989. ISBN: 0262031418.

[24] do Carmo, M. *Differential Geometry of Curves and Surfaces*. Pearson Education Canada, 1976. ISBN: 9780132125895.

[25] Dufourd, J.-F., Mathis, P., and Schreck, P. Geometric construction by assembling solved subfigures. *Artificial Intelligence 99*, 1 (Feb. 1998), 73–119. http://dx.doi.org/10.1016/S0004-3702(97)00070-2.

[26] Durand, C. B. *Symbolic and numerical techniques for constraint solving*. PhD thesis, Computer Science Department, Purdue University, 1998. Major Professor-C. M. Hoffmann.

[27] Elber, G., and Kim, M.-S. Geometric constraint solver using multivariate rational spline functions. In *Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications* (New York, NY, USA, 2001), SMA '01, ACM, pp. 1–10. http://dx.doi.org/10.1145/376957.376958.

[28] Even, S. *Graph Algorithms*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN: 0716780445.

[29] Fabre, A., and Schreck, P. Combining symbolic and numerical solvers to simplify indecomposable systems solving. In *Proceedings of the 2008 ACM Symposium on Applied Computing* (New York, NY, USA, 2008), SAC '08, ACM, pp. 1838–1842. http://dx.doi.org/10.1145/1363686.1364128.

[30] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN: 0-201-12110-7.

[31] Freeman-Benson, B. N., Maloney, J., and Borning, A. An incremental constraint solver. *Communications of the ACM 33*, 1 (Jan. 1990), 54–63. http://dx.doi.org/10.1145/76372.77531.

[32] Freixas, M., Joan-Arinyo, R., and Soto-Riera, A. A constraint-based dynamic geometry system. *Computer-Aided Design 42*, 2 (Feb. 2010), 151–161.

[33] Fudos, I. *Constraint solving for computer-aided design*. PhD thesis, Computer Science Department, Purdue University, West Lafayette, IN, USA, 1995.

[34] Fudos, I., and Hoffmann, C. M. Correctness proof of a geometric constraint solver. *International Journal of Computational Geometry and Applications 06*, 04 (1996), 405–420. http://dx.doi.org/10.1142/S0218195996000253.

[35] Fudos, I., and Hoffmann, C. M. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics 16*, 2 (1997), 179–216. http://dx.doi.org/10.1145/248210.248223.

[36] Gao, X.-S., Hoffmann, C. M., and Yang, W.-Q. Solving spatial basic geometric constraint configurations with locus intersection. In *Proceedings of the seventh ACM symposium on Solid modeling and applications* (New York, NY, USA, 2002), SMA '02, ACM, pp. 95–104. http://dx.doi.org/10.1145/566282.566299.

[37] Gao, X.-S., Huang, L.-D., and Jiang, K. A hybrid method for solving geometric constraint problems. In *Automated Deduction in Geometry'00* (2000), pp. 16–25. http://dx.doi.org/10.1007/3-540-45410-1_2.

[38] Gao, X.-S., Lin, Q., and Zhang, G.-F. A c-tree decomposition algorithm for 2d and 3d geometric constraint solving. *Computer-Aided Design 38*, 1 (2006), 1 – 13. http://dx.doi.org/10.1016/j.cad.2005.03.002.

[39] Gross, J. L., and Yellen, J. *Graph Theory and Its Applications, Second Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2005. ISBN: 158488505X.

[40] Haller, K., John, A. L.-S., Sitharam, M., Streinu, I., and White, N. Body-and-cad geometric constraint systems. In *Proceedings of the 2009 ACM symposium on Applied Computing* (New York, NY, USA, 2009), SAC '09, ACM, pp. 1127–1131. http://dx.doi.org/10.1145/1529282.1529530.

[41] Hel-Or, Y., Rappoport, A., and Werman, M. Relaxed parametric design with probabilistic constraints. In *Proceedings on the Second ACM Symposium on Solid Modeling and Applications* (New York, NY, USA, 1993), SMA '93, ACM, pp. 261–270. http://dx.doi.org/10.1145/164360.164437.

[42] Heydon, A., and Nelson, G. *The Juno-2 constraint-based drawing editor*. SRC reports. Digital, Systems Research Center, 1994.

[43] Hillyard, R. C., and Braid, I. C. Characterizing non-ideal shapes in terms of dimensions and tolerances. *ACM SIGGRAPH Computer Graphics 12*, 3 (Aug. 1978), 234–238. http://dx.doi.org/10.1145/965139.807396.

[44] Hoffman, C. M., Lomonosov, A., and Sitharam, M. Decomposition plans for geometric constraint problems, part ii: New algorithms. *Journal of Symbolic Computation 31*, 4 (2001), 409 – 427. http://dx.doi.org/10.1006/jsco.2000.0403.

[45] Hoffman, C. M., Lomonosov, A., and Sitharam, M. Decomposition plans for geometric constraint systems, part i: Performance measures for CAD. *Journal of Symbolic Computation 31*, 4 (2001), 367 – 408. http://dx.doi.org/10.1006/jsco.2000.0402.

[46] Hoffmann, C., Lomonosov, A., and Sitharam, M. Finding solvable subsets of constraint graphs. In *Principles and Practice of Constraint Programming-CP97*, G. Smolka, Ed., vol. 1330 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1997, pp. 463–477. http://dx.doi.org/10.1007/BFb0017460.

[47] Hoffmann, C., and Vermeer, P. *A spatial constraint problem*. Springer Netherlands, Nice, France, Sept 1995. http://dx.doi.org/10.1007/978-94-011-0333-6_9.

[48] Hoffmann, C. M., and Joan-Arinyo, R. Symbolic constraints in constructive geometric constraint solving. *Journal of Symbolic Computation 23*, 2-3 (1997), 287–299. http://dx.doi.org/10.1006/jsco.1996.0089.

[49] Hoffmann, C. M., and Joan-Arinyo, R. A brief on constraint solving. *Computer-Aided Design and Applications 5*, 2 (2005), 655–663. http://dx.doi.org/10.1080/16864360.2005.10738330.

[50] Hoffmann, C. M., Sitharam, M., and Yuan, B. Making constraint solvers more usable: overconstraint problem. *Computer-Aided Design 36*, 4 (2004), 377 – 399. http://dx.doi.org/10.1016/S0010-4485(03)00099-X.

[51] Hopcroft, J., and Tarjan, R. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM 16*, 6 (June 1973), 372–378. http://dx.doi.org/10.1145/362248.362272.

[52] Hsu, C.-Y. *Graph-based Approach for Solving Geometric Constraint Problems*. PhD thesis, University of Utah Salt Lake City, Salt Lake City, UT, USA, 1996. UMI Order No. GAX96-30587.

[53] Hsu, C.-Y., and Brüderlin, B. A hybrid constraint solver using exact and iterative geometric constructions. In *CAD Systems Development: Tools and Methods [Dagstuhl Seminar, 1995]* (London, UK, 1997), Springer-Verlag, pp. 265–279.

[54] Jackson, B., and Jordán, T. Globally rigid circuits of the direction–length rigidity matroid. *Journal of Combinatorial Theory, Series B 100*, 1 (2010), 1 – 22. http://dx.doi.org/10.1016/j.jctb.2009.03.004.

[55] Jermann, C. *Résolution de contraintes géométriques par rigidification récursive et propagation d'intervalles*. PhD thesis, Université de Nice Sophia-Antipolis, Dec. 2002.

[56] Jermann, C., and Hosobe, H. A constraint hierarchies approach to geometric constraints on sketches. In *Proceedings of the 2008 ACM symposium on Applied computing* (New York, NY, USA, 2008), SAC '08, ACM, pp. 1843–1844. http://dx.doi.org/10.1145/1363686.1364130.

[57] Jermann, C., Neveu, B., and Trombettoni, G. Algorithms for identifying rigid subsystems in geometric constraint systems. In *International Joint Conference on Artificial Intelligence IJCAI 2003* (2003), vol. 18, pp. 233–238.

[58] Jermann, C., Trombettoni, G., Neveu, B., and Mathis, P. Decomposition of geometric constraint systems: a survey. *International Journal of Computational Geometry and Applications 16*, 5-6 (2006), 379–414. http://dx.doi.org/10.1142/S0218195906002105.

[59] Joan-Arinyo, R. Basics on geometric constraint solving. In *EGC'09: Encuentros de Geometria Computacional. Oral presentation* (2009).

[60] Joan-Arinyo, R., and Soto, A. A correct rule-based geometric constraint solver. *Computers & Graphics 21*, 5 (1997), 599 – 609. http://dx.doi.org/10.1016/S0097-8493(97)00038-1.

[61] Joan-Arinyo, R., Soto, A., and Vila, S. Resolución de restricciones geométricas. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial 20* (2003), 121–136.

[62] 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. http://dx.doi.org/10.1145/300776.300780.

[63] Joan-Arinyo, R., Soto-Riera, A., Tarrés-Puertas, M., and Vila-Marta, S. Decomposition of geometric constraint graphs based on computing fundamental circuits. Research Report LSI-07-31-R, Software Department, Technical University of Catalunya, 2007.

[64] Joan-Arinyo, R., Soto-Riera, A., and Vila-Marta, S. Constraint-based techniques to support collaborative design. *Journal of Computing and Information Science in Engineering 6* (2006), 139–148.

[65] 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 17th Spring Conference on Computer Graphics (SCCG '01)* (Los Alamitos, CA, USA, 2001), IEEE Computer Society, pp. 49 –54. http://doi.ieeecomputersociety.org/10.1109/SCCG.2001.945336.

[66] Joan-Arinyo, R., Soto-Riera, A., Vila-Marta, S., and Vilaplana, J. Declarative characterization of a general architecture for constructive geometric constraint solvers. In *The Fifth International Conference on Computer Graphics and Artificial Intelligence* (Limoges, France, 14-15 May 2002), D. Plemenos, Ed., Université de Limoges, pp. 63–76.

[67] 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 *Proceedings of the eighth ACM symposium on Solid modeling and applications* (New York, NY, USA, 2003), SM '03, ACM, pp. 33–44. http://dx.doi.org/10.1145/781606.781616.

[68] Joan-Arinyo, R., Soto-Riera, A., Vila-Marta, S., and Vilaplana-Pastó, J. Revisiting decomposition analysis of geometric constraint graphs. *Computer-Aided Design 36*, 2 (2004), 123 – 140. http://dx.doi.org/10.1016/S0010-4485(03)00057-5.

[69] Joan-Arinyo, R., Tarrés-Puertas, M., and Vila-Marta, S. Geometric constraint graphs decomposition based on computing graph circuits. In *Automated Deduction in Geometry - 7th International Workshop, ADG 2008, Shanghai: East China Normal University, China* (2008), pp. 96–101.

[70] Joan-Arinyo, R., Tarrés-Puertas, M., and Vila-Marta, S. Treedecomposition of geometric constraint graphs based on computing graph circuits. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling* (New York, NY, USA, 2009), SPM '09, ACM, pp. 113–122. http://dx.doi.org/10.1145/1629255.1629270.

[71] Joan-Arinyo, R., Tarrés-Puertas, M., and Vila-Marta, S. Decomposition of geometric constraint graphs based on computing fundamental circuits. Correctness and complexity. *Computer-Aided Design 52* (2014), 1–16. http://dx.doi.org/10.1016/j.cad.2014.02.006.

[72] Johnson, L. W., and Riess, R. *Numerical analysis*, second ed. Addison-Wesley, 1982. ISBN: 9780201103922.

[73] Kassimali, A. *Structural Analysis*. Cengage Learning, 2010. ISBN: 9780495295655.

[74] Kondo, K. Pigmod: parametric and interactive geometric modeller for mechanical design. *Computer-Aided Design 22*, 10 (Jan. 1991), 633–644. http://dx.doi.org/10.1016/0010-4485(90)90010-A.

[75] Kondo, K. Algebraic method for manipulation of dimensional relationships in geometric models. *Computer-Aided Design 24*, 3 (1992), 141 – 147. http://dx.doi.org/10.1016/0010-4485(92)90033-7.

[76] Kramer, G. *Solving Geometric Constraint Systems: A Case Study in Kinematics*. Artificial Intelligence Series. Mit Press, 1992. ISBN: 9780262111645.

[77] Kramer, G. A. Solving geometric constraint systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 1* (1990), AAAI'90, AAAI Press, pp. 708–714.

[78] Kramer, G. A. Using degrees of freedom analysis to solve geometric constraint systems. In *Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications* (New York, NY, USA, 1991), SMA '91, ACM, pp. 371–378. http://dx.doi.org/10.1145/112515.112566.

[79] Kramer, G. A. A geometric constraint engine. *Artificial Intelligence 58* (1992), 327 – 360. http://dx.doi.org/10.1016/0004-3702(92)90012-M.

[80] Kwaiter, G., Gaildrat, V., and Caubet, R. Interactive constraint system for solid modeling objects. In *Proceedings of the fourth ACM symposium on Solid modeling and applications* (New York, NY, USA, 1997), SMA '97, ACM, pp. 265–270. http://dx.doi.org/10.1145/267734.267803.

[81] Laman, G. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics 4* (1970), 331–340. http://dx.doi.org/10.1007/BF01534980.

[82] Lamure, H., and Michelucci, D. Solving geometric constraints by homotopy. In *Proceedings of the third ACM symposium on Solid modeling and applications* (New York, NY, USA, 1995), SMA '95, ACM, pp. 263–269. http://dx.doi.org/10.1145/218013.218071.

[83] Latham, R., and Middleditch, A. Connectivity analysis: a tool for processing geometric constraints. *Computer-Aided Design 28*, 11 (1996), 917 – 928. http://dx.doi.org/10.1016/0010-4485(96)00023-1.

[84] Lee, J. Y., and Kim, K. Geometric reasoning for knowledge-based parametric design using graph representation. *Computer-Aided Design 28*, 10 (1996), 831 – 841. http://dx.doi.org/10.1016/0010-4485(96)00016-4.

[85] Lee, K.-Y., Kwon, O.-H., Lee, J.-Y., and Kim, T.-W. A hybrid approach to geometric constraint solving with graph analysis and reduction. *Adv. Eng. Softw. 34*, 2 (Feb. 2003), 103–113. http://dx.doi.org/10.1016/S0965-9978(02)00108-4.

[86] Leler, W. *Constraint programming languages: their specification and generation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. ISBN: 0-201-06243-7.

[87] Light, R., and Gossard, D. Modification of geometric models through variational geometry. *Computer-Aided Design 14*, 4 (1982), 209 – 214. http://dx.doi.org/10.1016/0010-4485(82)90292-5.

[88] Lin, V. C., Gossard, D. C., and Light, R. A. Variational geometry in computer-aided design. *SIGGRAPH Comput. Graph. 15*, 3 (Aug. 1981), 171–177. http://dx.doi.org/10.1145/965161.806803.

[89] Lomonosov, A. *Graph and combinatorial algorithms for geometric constraint solving.* PhD thesis, University of Florida, Gainesville, FL, USA, 2004.

[90] Luo, Z., Hu, W., and Feng, E. Computing curve intersection by homotopy methods. *J. Comput. Appl. Math. 236*, 5 (Oct. 2011), 892–905. http://dx.doi.org/10.1016/j.cam.2011.05.015.

[91] Michelucci, D., Schreck, P., Thierry, S. E. B., Fünfzig, C., and Génevaux, J.-D. Using the witness method to detect rigid subsystems of geometric constraints in cad. In *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling* (New York, NY, USA, 2010), SPM '10, ACM, pp. 91–100. http://dx.doi.org/10.1145/1839778.1839791.

[92] Miller, G. L., and Ramachandran, V. A new graph triconnectivity algorithm and its parallelization. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (New York, 1987), ACM, pp. 335–344. http://dx.doi.org/10.1145/28395.28431.

[93] Miller, G. L., and Ramachandran, V. A new graph triconnectivity algorithm and its parallelization. *Combinatorica 12*, 1 (1992), 53–76.

[94] Murota, K. *Systems Analysis by Graphs and Matroids: Structural Solvability and Controllability (Algorithms and Combinatorics, Vol 3).* Springer, July 1987. ISBN: 0387176594.

[95] Mutzel, P., and Weiskircher, R. Computing optimal embeddings for planar graphs. In *Proceedings COCOON '00, volume 1858 of LNCS* (2000), Springer Verlag, pp. 95–104.

[96] Nishizeki, T., and N.Chiba. *Planar Graphs. Theory and Algorithms*, first ed. Dover Publications, Inc., Mineola, New York, 2008. ISBN: 9780080867748.

[97] Nishizeki, T., and Rahman, M. *Planar Graph Drawing*. Lecture Notes Series on Computing Series. World Scientific, 2004. ISBN: 9789812560339.

[98] Owen, J. Constraint on simple geometry in two and three dimensions. *International Journal of Computational Geometry & Applications 06*, 04 (1996), 421–434. http://dx.doi.org/10.1142/S0218195996000265.

[99] Owen, J. C. Algebraic solution for geometry from dimensional constraints. In *Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications* (New York, NY, USA, 1991), SMA '91, ACM, pp. 397–407. http://dx.doi.org/10.1145/112515.112573.

[100] Podgorelec, D. A new constructive approach to constraint-based geometric design. *Computer-Aided Design 34*, 11 (2002), 769 – 785. http://dx.doi.org/10.1016/S0010-4485(01)00133-6.

[101] Reif, J. H. Depth-first search is inherently sequential. *Inf. Process. Lett. 20*, 5 (1985), 229–234. http://dx.doi.org/10.1016/0020-0190(85)90024-9.

[102] Salomons, O. W., van Slooten, F., van Houten, F. J. A. M., and Kals, H. J. J. Conceptual graphs in constraint based re-design. In *Proceedings of the third ACM symposium on Solid modeling and applications* (New York, NY, USA, 1995), SMA '95, ACM, pp. 55–64. http://dx.doi.org/10.1145/218013.218030.

[103] Sannella, M. Skyblue: a multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th annual ACM symposium on User interface software and technology* (New York, NY, USA, 1994), UIST '94, ACM, pp. 137–146. http://dx.doi.org/10.1145/192426.192485.

[104] Savage, J. E. *Models of Computation: Exploring the Power of Computing*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN: 0201895390.

[105] Schreck, P., and Schramm, i. Using invariance under the similarity group to solve geometric constraint systems. *Comput. Aided Des. 38*, 5 (May 2006), 475–484. http://dx.doi.org/10.1016/j.cad.2006.01.002.

[106] Serrano, D. Automatic dimensioning in design for manufacturing. In *SMA '91: Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications* (New York, NY, USA, 1991), ACM, pp. 379–386. http://dx.doi.org/10.1145/112515.112568.

[107] Servatius B., S. H., and J., G. *Combinatorial rigidity*. Graduate Studies in Mathematics. American Mathematical Society, 1993. ISBN: 9780821838013.

[108] Sitharam, M. *Combinatorial approaches to geometric constraint solving: Problems, progress and directions*. DIMACS Series in Discrete Mathematics and Theoretical Computer Sicience, 2005. ISBN: 9780821836286.

[109] Sitharam, M., Arbree, A., Zhou, Y., and Kohareswaran, N. Solution space navigation for geometric constraint systems. *ACM Transactions on Graphics 25*, 2 (Apr. 2006), 194–213. http://dx.doi.org/10.1145/1138450.1138452.

[110] Sitharam, M., Oung, J.-J., Zhou, Y., and Arbree, A. Geometric constraints within feature hierarchies. *Computer-Aided Design 38*, 1 (Jan. 2006), 22–38. http://dx.doi.org/10.1016/j.cad.2005.05.001.

[111] Sitharam, M., Zhou, Y., and Peters, J. Reconciling conflicting combinatorial preprocessors for geometric constraint systems. *International Journal of Computational Geometry and Applications 20*, 6 (2010), 631–651. http://dx.doi.org/10.1142/S0218195910003463.

[112] Solano, L., and Brunet, P. Constructive constraint-based model for parametric cad systems. *Computer-Aided Design 26*, 8 (1994), 614 – 621. http://dx.doi.org/10.1016/0010-4485(94)90104-X.

[113] Soto-Riera, A. *Geometric Constraint Solving in 2D*. PhD thesis, Departament de Llenguatges i Sistemes Informàtics, UPC, 1998.

[114] Soto-Riera, A., Vila-Marta, S., Silva, D., and Freixa, M. The SolBCN geometric constraint solver. Source code from http://floss.lsi.upc.edu, 2007. Under GNU-GPL license.

[115] Sridhar, N., Agrawal, R., and Kinzel, G. L. Algorithms for the structural diagnosis and decomposition of sparse, underconstrained design systems. *Computer-Aided Design 28*, 4 (1996), 237–249. http://dx.doi.org/10.1016/0010-4485(96)88488-0.

[116] Sunde, G. A CAD system with declarative specification of shape. *Eurographics Workshop on Intelligent CAD Systems* (April 1987), 90–105.

[117] Sutherland, I. E. *Seminal graphics: poineering efforts that shaped the field*. ACM, New York, NY, USA, 1998, ch. Sketchpad—a man-machine graphical communication system, pp. 391–408. http://doi.acm.org/10.1145/280811.281031.

[118] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*, 2 (1972), 146–160. http://dx.doi.org/10.1137/0201010.

[119] Tarrés-Puertas, M. Direct tree decomposition of geometric constraint graphs. Research report, Software Department, Technical University of Catalunya, 2008.

[120] Thierry, S. *Décomposition et paramétrisation de systèmes de contraintes géométriques sous-contraints*. PhD thesis, Université de Strasbourg, Sep 2010.

[121] Thierry, S. E. B. A particle-spring approach to geometric constraints solving. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (New York, NY, USA, 2011), SAC '11, ACM, pp. 1100–1105. http://dx.doi.org/10.1145/1982185.1982428.

[122] Thierry, S. E. B., Mathis, P., and Schreck, P. Towards an homogeneous handling of under-constrained and well-constrained systems of geometric constraints. In *Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), SAC '07, ACM, pp. 773–777. http://dx.doi.org/10.1145/1244002.1244174.

[123] Thulasiraman, K., and Swamy, M. N. S. *Graphs: Theory and Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 1992. ISBN: 0471513563.

[124] Todd, P. A k-tree generalization that characterized consistency of dimensioned engineering drawings. *SIAM Journal on Discrete Mathematics 2*, 2 (May 1989), 255–261. http://dx.doi.org/10.1137/0402022.

[125] Veltkamp, R., and Arbab, F. Geometric constraint propagation with quantum labels. In *Eurographics Workshop on Computer Graphics and Mathematics* (1992), B. Falcidieno, I. Herman, and C. Pienovi, Eds., Eurographics, Springer-Verlag, pp. 211–228.

[126] Verroust, A. *Etude de Problemes Lies a la Definition, la Visualisation et l'Animation d'Objects Complexes en Informatique Graphique*. PhD thesis, Universite de Paris-Sud, Centre d'Orsay, 1990.

[127] Verroust, A., Schonek, F., and Roller, D. Rule-oriented method for parameterized computer-aided design. *Computer-Aided Design 24*, 10 (1992), 531 – 540. http://dx.doi.org/10.1016/0010-4485(92)90040-H.

[128] Vila-Marta, S. *Contribution to Geometric Constraint Solving in Cooperative Engineering*. PhD thesis, Departament de Llenguatges i Sistemes Informàtics, UPC, 2003.

[129] Voss, H. *Cycles and Bridges in Graphs*. Mathematics and its applications (Kluwer Academic Publishers).: East European Series. Springer, 1991. ISBN: 9780792308997.

[130] Wen-Tsun, W. Basic principles of mechanical theorem proving in elementary geometries. *Journal of Automated Reasoning 2* (1986), 221–252. http://dx.doi.org/10.1007/BF02328447.

[131] Wen-Tsün, W. *Mechanical theorem proving in geometries - basic principles*. Texts and monographs in symbolic computation. Springer, 1994. ISBN: 9783211825068.

[132] Wen-Tsün, W. Automatic geometry theorem-proving and automatic geometry problem-solving. In *Proceedings of the Second International Workshop on Automated Deduction in Geometry* (London, UK, UK, 1999), ADG '98, Springer-Verlag, pp. 1–13. http://dl.acm.org/citation.cfm?id=647692.731041.

[133] Whiteley, W. *Handbook of discrete and computational geometry*. CRC Press, Inc., Boca Raton, FL, USA, 1997, ch. Rigidity and scene analysis, pp. 893–916. ISBN: 0849385245.

[134] Yamaguchi, Y., and Kimura, F. A constraint modeling system for variational geometry. In *Geometric Modeling for Product Engineering*, J. U. Turner, M. J. Wozny, and K. Preiss, Eds. Elsevier North Holland, 1990, pp. 221–233.

[135] Yeguas, E., Joan-Arinyo, R., and Victoria Luzón, M. Modeling the performance of evolutionary algorithms on the root identification problem: A case study with pbil and chc algorithms. *Evolutionary Computation 19*, 1 (Mar. 2011), 107–135. http://dx.doi.org/10.1162/EVCO_a_00017.

[136] Zhang, G.-F. Well-constrained completion for under-constrained geometric constraint problem based on connectivity analysis of graph. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (New York, NY, USA, 2011), SAC '11, ACM, pp. 1094–1099. http://dx.doi.org/10.1145/1982185.1982427.

[137] Zhou, Y. *Combinatorial decomposition, generic independence and algebraic complexity of geometric constraints systems: applications in biology and engineering.* PhD thesis, University of Florida, Gainesville, FL, USA, 2006.

Case Study Decomposition Examples

1 Case Study Decomposition: A 2D cross section.

To further illustrate how the approach works, we consider now a $2D$ cross section that will be used to model a $3D$ shape by sweeping it along a given trajectory. The $2D$ cross section depicted in Figure 1 is adapted from [83].

Our approach only considers constraint problems where geometric elements have two degrees of freedom. Therefore, circles and arcs have fixed radii and what needs to be placed with respect to other geometries is just their centers. In these conditions, the $2D$ cross section includes 14 points and 11 segments. Constraints include point-point distances, perpendicular point-segment distances and segment-segment angles. They are depicted in Figure 1 as thin lines. Distance constraints are depicted as straight segments with arrows, arbitrary angle constraints are depicted as arcs of circle while 90° angles are depicted as a pair of perpendicular straight segments with a common bound.

The constraint graph is shown in Figure 2. Geometric elements are depicted as the graph vertices. Constraints are translated as graph edges. Since the problem is well-constrained, with $|V| = 25$ and $|E| = 47$, the Laman relation, $|E| = 2|V| - 3$, holds. It is routine to check that for each subgraph, $|E' \leq 2|V'| - 3$ also holds.

As explained in Section 4.4.1 in a preprocessing step, our approach trivially solves constructions corresponding to geometric elements whose nodes in the constraint graph have degree two, that is, geometric elements on which just two constraints have been defined. After sequentially removing degree two vertices, the connectivity of the graph to be further considered is two or higher. The graph is shown in Figure 3.

Assume that the spanning tree to the constraint graph computed by the algorithm is the one depicted in Figure 4 starting in arbitrary root p_5.

1.1 Computing the Set of Fundamental Circuits

The set of fundamental circuits induced by the spanning tree in Figure 4 is shown in Figure 5. The fundamental circuits contain the following vertices:

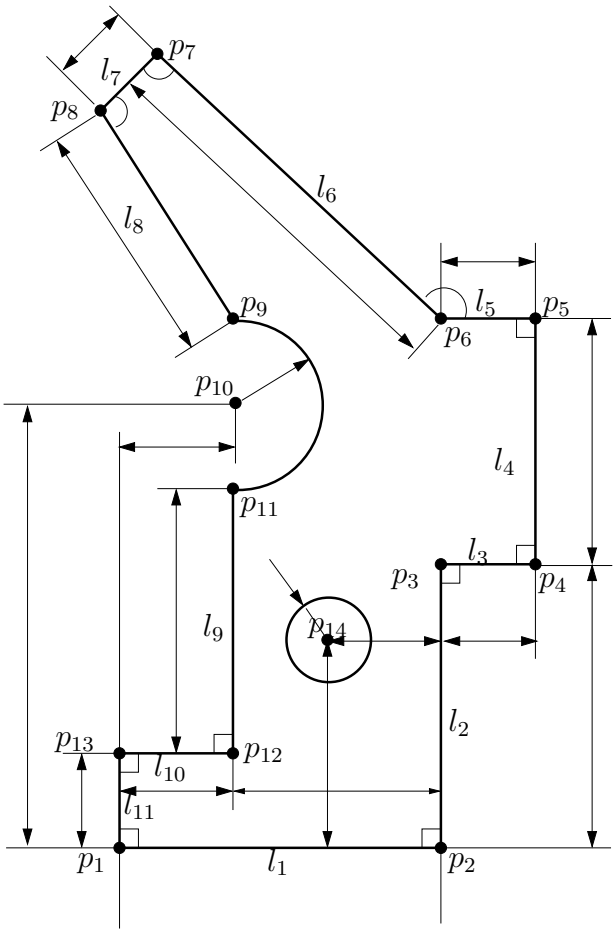


Figure 1: $2D$ cross section parametrically defined by geometric constraints.

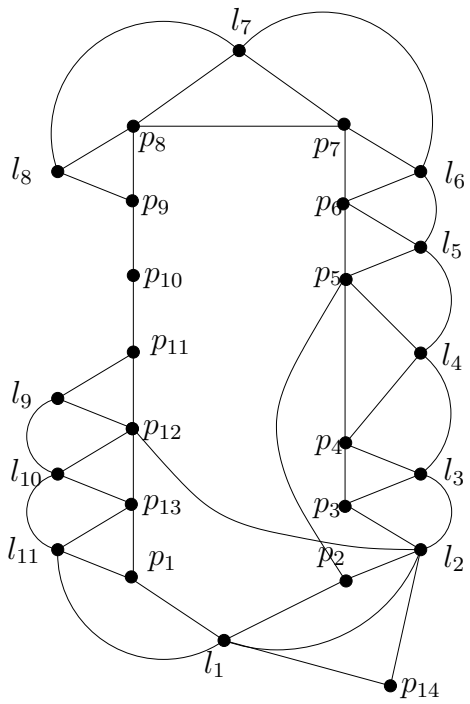


Figure 2: Constraint Graph for the $2D$ cross section in Figure 1.

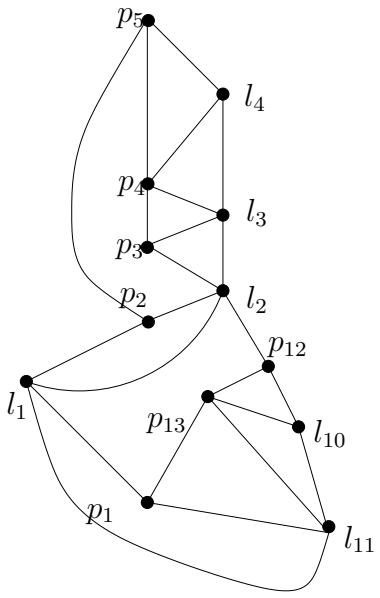


Figure 3: Geometric constraint graph for the $2D$ cross section in Figure 1 after sequentially solving geometric elements supporting just two constraints.

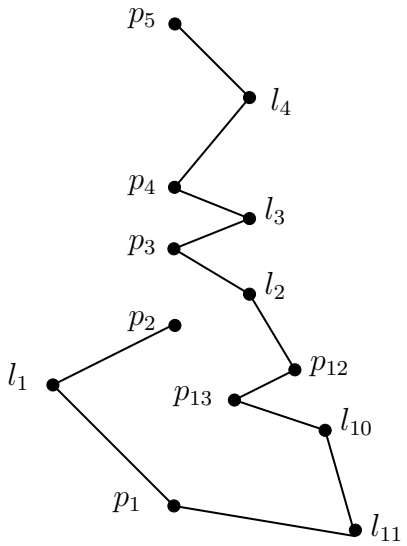


Figure 4: Spanning Tree T of G.

$$
\begin{aligned}
C_1 &= \langle p_5, l_4, p_4, l_3, p_3, l_2, p_{12}, p_{13}, l_{10}, l_{11}, p_1, l_1, p_2 \rangle \\
C_2 &= \langle p_5, l_4, p_4 \rangle \\
C_3 &= \langle l_4, p_4, l_3 \rangle \\
C_4 &= \langle p_4, l_3, p_3 \rangle \\
C_5 &= \langle l_3, p_3, l_2 \rangle \\
C_6 &= \langle l_2, p_{12}, p_{13}, l_{10}, l_{11}, p_1, l_1, p_2 \rangle \\
C_7 &= \langle l_2, p_{12}, p_{13}, l_{10}, l_{11}, p_1, l_1 \rangle \\
C_8 &= \langle p_{13}, l_{10}, l_{11}, p_1 \rangle \\
C_9 &= \langle p_{12}, p_{13}, l_{10} \rangle \\
C_{10} &= \langle p_{13}, l_{10}, l_{11} \rangle \\
C_{11} &= \langle l_{11}, p_1, l_1 \rangle
\end{aligned}
$$

1.2 Computing the Set of Bridges

We choose an arbitrary fundamental circuit from the graph G, for example, $C_6 = \langle l_2, p_{12}, p_{13}, l_{10}, l_{11}, p_1, l_1, p_2 \rangle$, and compute the set of bridges following the approach in Chapter 4.4.5.

Let $S \subset G$ be the graph defined by the vertices and edges corresponding to the fundamental circuit C_6, that is $S = (\{l_2, p_{12}, p_{13}, l_{10}, l_{11}, p_1, l_1, p_2\}, \{(l_2, p_{12}), (p_{12}, p_{13}), (p_{13}, l_{10}), (l_{10}, l_{11}), (l_{11}, p_1), (p_1, l_1), (l_1, p_2)\})$. The dashed circle in Figure 6 shows the class κ_1 defined by the circuit C_6, depicted in thick line, in the graph example.

S induces on G a set of components: one non-singular component and the rest are singular components. The set of components is shown in Figure 7.

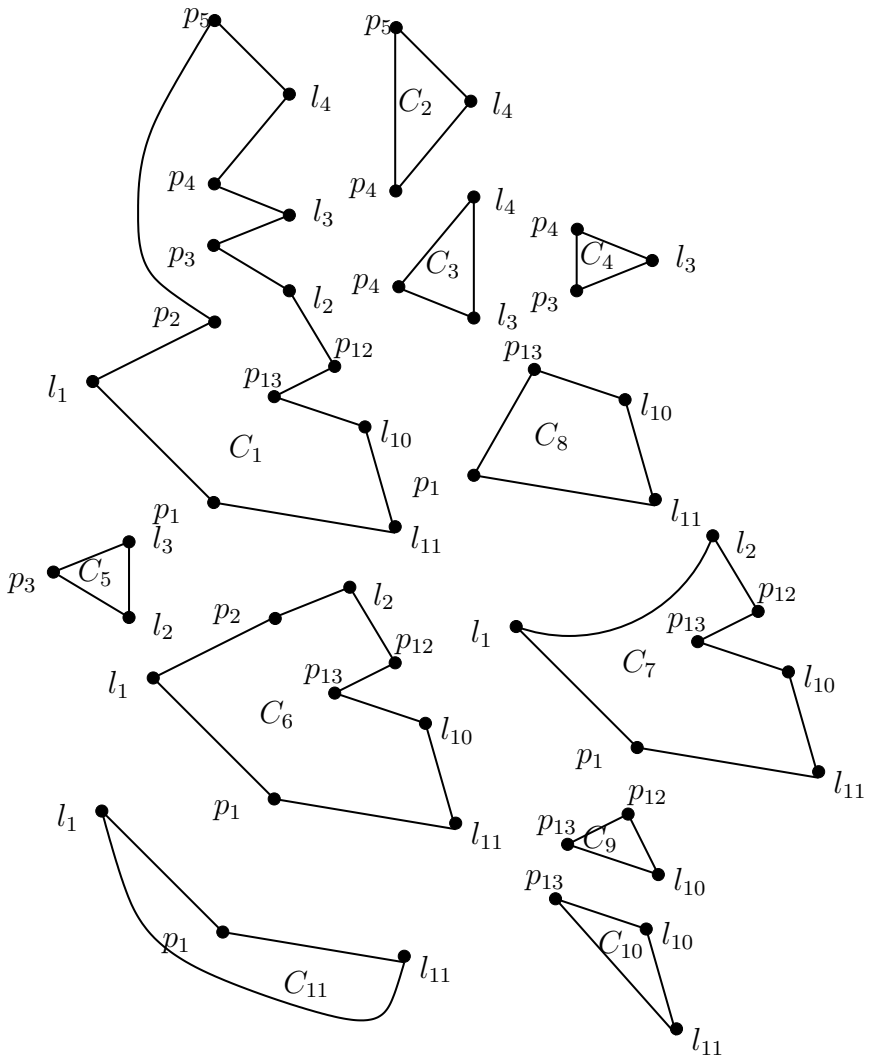


Figure 5: Fundamental circuits in the graph associated to the spanning tree depicted in Figure 4.

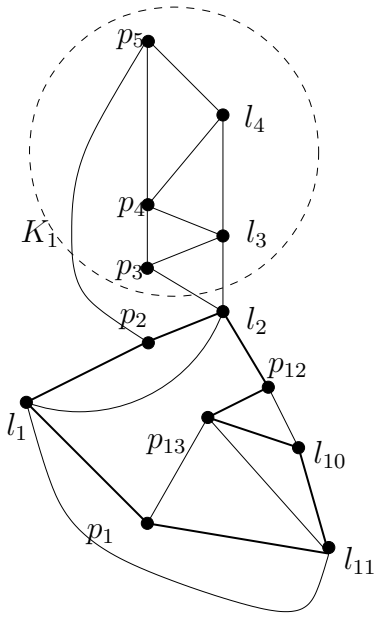


Figure 6: Classes induced by the circuit C_6, (thick line) in the graph.

The set of bridges in the graph corresponding to the components in Figure 7 is depicted in Figure 8 and are the following,

$B_1 = (\{p_2, p_5, p_4, l_4, l_3, p_3, l_2\}, \{(p_5, p_2), (p_5, l_4), (p_5, p_4), (l_4, p_4), (l_4, l_3), (p_4, l_3), (p_4, p_3), (p_3, l_3), (l_3, l_2), (p_3, l_2)\})$,

$$\begin{aligned} B_2 &= (\{p_{12}, l_{10}\}, \{(p_{12}, l_{10})\}) \\ B_3 &= (\{p_1, p_{13}\}, \{(p_1, p_{13})\}) \\ B_4 &= (\{p_{13}, l_{11}\}, \{(p_{13}, l_{11})\}) \\ B_5 &= (\{l_1, l_{11}\}, \{(l_1, l_{11})\}) \\ B_6 &= (\{l_1, l_2\}, \{(l_1, l_2)\}) \end{aligned}$$

1.3 Computing the Collapsed graph

Following the procedure explained in Chapter 4.4.6, bridges are represented by means of a star graph. Figure 9 shows the collapsed graph G' corresponding to graph G after replacing every bridge H_i by a star B_i, and where each bridge is identified by its center vertex.

1.4 Computing the Merged Graph

We proceed to merge bridges of G'. Bridges B_3 and B_5 have interlaced attachments and are merged into B_{35}. Bridges B_2, B_{35}, B_4 have interlaced and are merged into B_{2345}. The resulting merged graph of G', named G'', is shown in Figure 10.

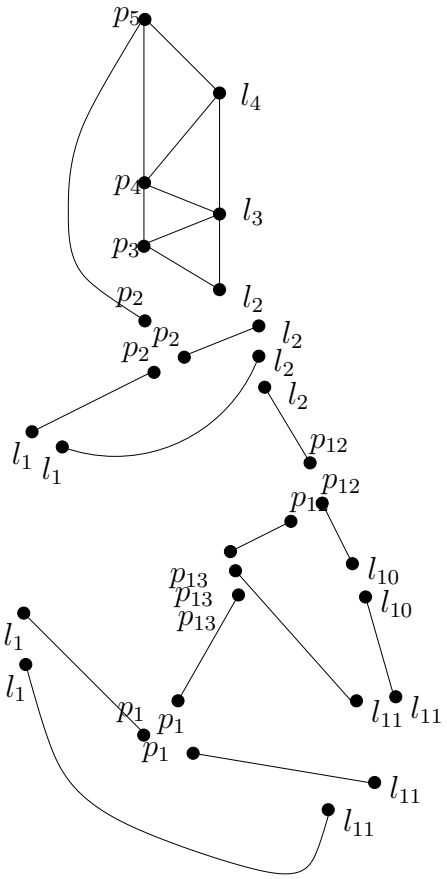


Figure 7: Components defined by the classes in the graph for the circuit C_6.

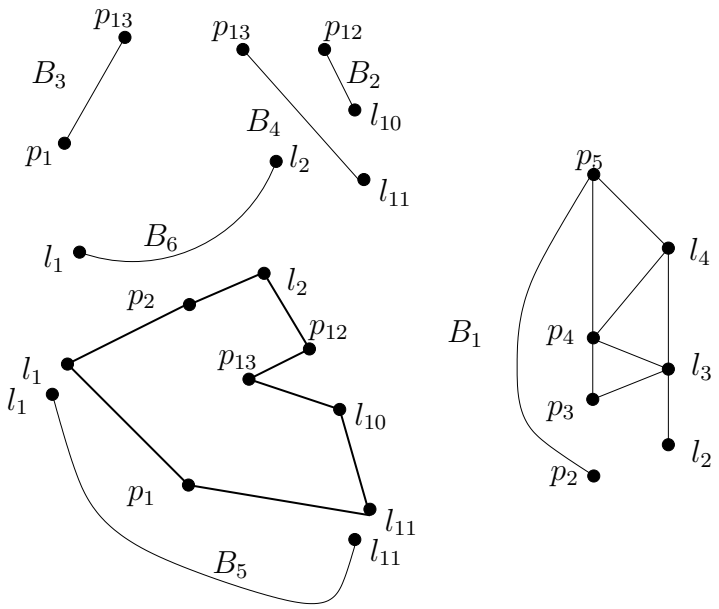


Figure 8: Bridges (thin line) induced in the constraint graph by circuit C_6.

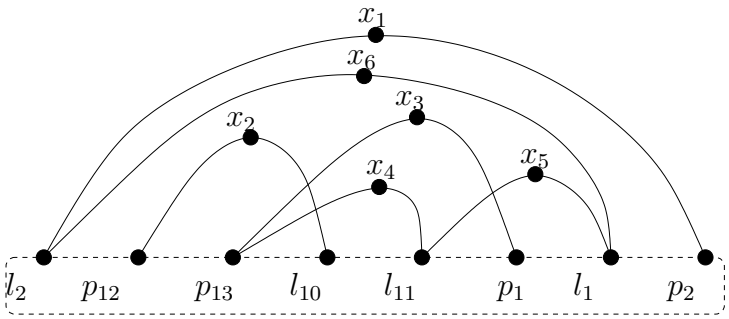


Figure 9: Collapsed Graph.

1.5 Computing the Planar Embedding of the Merged Graph

The next step is to compute a planar graph. Figure 11 depicts the planar graph G''' as the result of embedding the merged graph G''.

The set of faces defined by the planar embedding of G''' are the following.

$$
\begin{aligned}
f_u &= \{l_2, x_1, p_2\} & f_1 &= \{l_2, x_6, l_1, p_2\} \\
f_2 &= \{l_2, p_{12}, x_{2345}, l_1\} & f_3 &= \{p_{12}, x_{2345}, p_{13}\} \\
f_4 &= \{p_{13}, x_{2345}, l_{10}\} & f_5 &= \{l_{10}, x_{2345}, l_{11}\} \\
f_6 &= \{l_{11}, x_{2345}, p_1\} & f_7 &= \{p_1, x_{2345}, l_1\}
\end{aligned}
$$

1.6 Computing the Hinges

To compute a set of hinges, we search the set of faces $\{f_u, f_1, \ldots, f_9\}$. Up to two faces in the planar embedding which share three vertices with the bounded face f_b can be identified. The pair of faces (f_b, f_2) share vertices $\{l_2, p_{12}, l_1\}$ and the pair (f_b, f_1) share the vertices $\{l_2, l_1, p_2\}$. Therefore two different decompositions can be performed.

If we assume that the algorithm identifies first the triple of vertices $\{l_2, p_{12}, l_1\}$ corresponding to the pair of faces (f_b, f_2), the decomposition step yields the clusters G_1, G_2, G_3 as shown in Figure 12. Next the algorithm is recursively applied to graphs G_1 and G_3. Notice that G_2

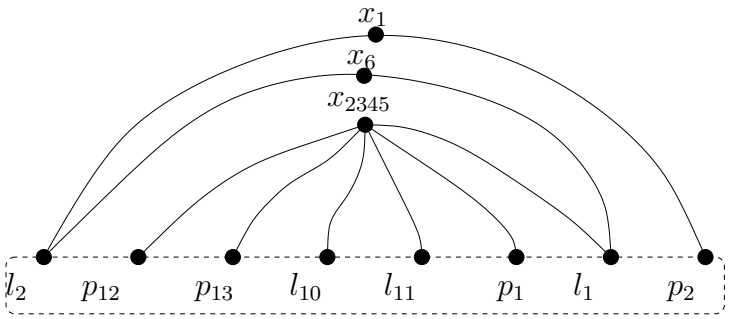


Figure 10: Merged Graph.

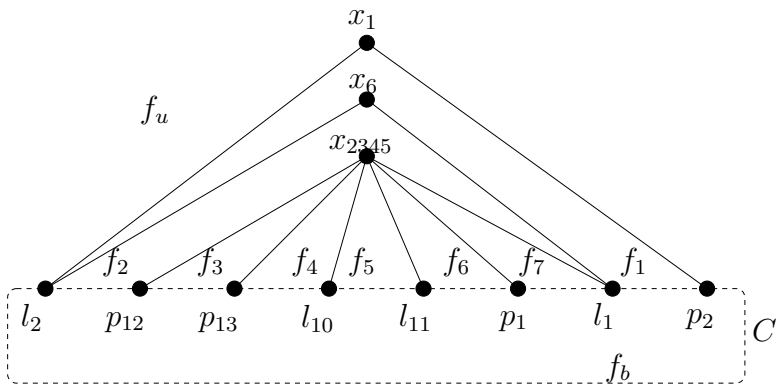


Figure 11: Planar embedding of the merged graph corresponding to the fundamental circuit C_6 in Figure 5.

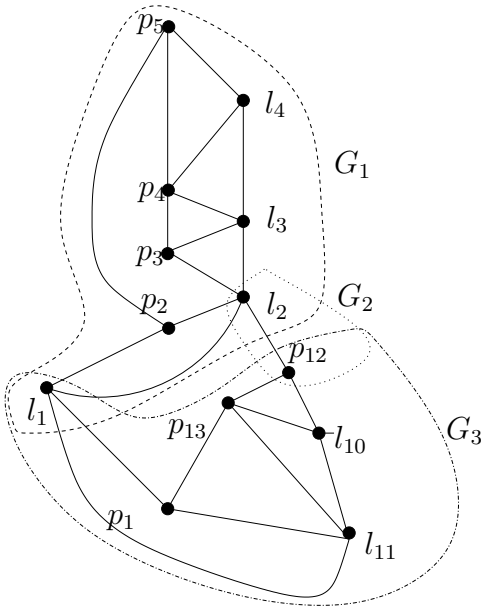


Figure 12: Clusters G_1, G_2 and G_3 defined by faces f_b and f_2 in the planar embedding of the $2D$ cross section merged graph.

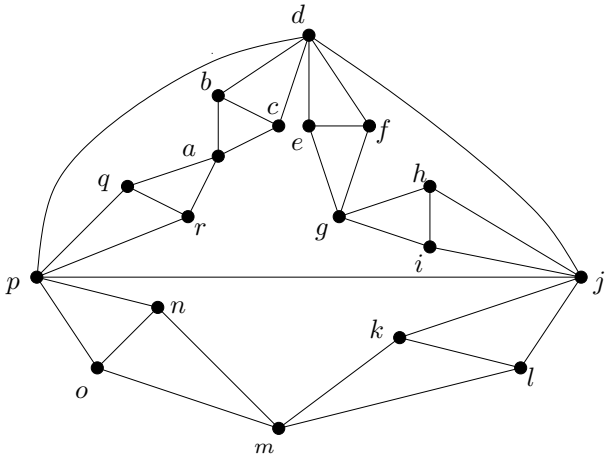


Figure 13: Graph G.

includes two geometries and one constraint on them therefore is a trivial geometry placing problem and no further decomposition is needed.

2 Case Study Decomposition: A theoretical example

In this Section we give a geometric constraint graph and the results yielded by our decomposition algorithm. Given a biconnected graph $G = (V, E)$, our algorithm finds hinges (if exist) and clusters in the graph. Figure 13 shows the constraint graph $G = (V, E)$ of study.

2.1 Computing the Set of Fundamental Circuits

We start computing the set of fundamental circuits of G, as we explained in section 4.4.4.

First, we apply a depth-first search to graph G, starting in the arbitrary root a. We obtain the tree shown in Figure 14 a).

The chords are the edges of Figure 13, not visited in the previous step. Every chord induces a fundamental circuit. Figure 14 b) shows the resulting DFS tree and the fundamental circuits (chords) $C_1, C_2, \ldots, C_{16}$. The fundamental circuits contain the following vertices:

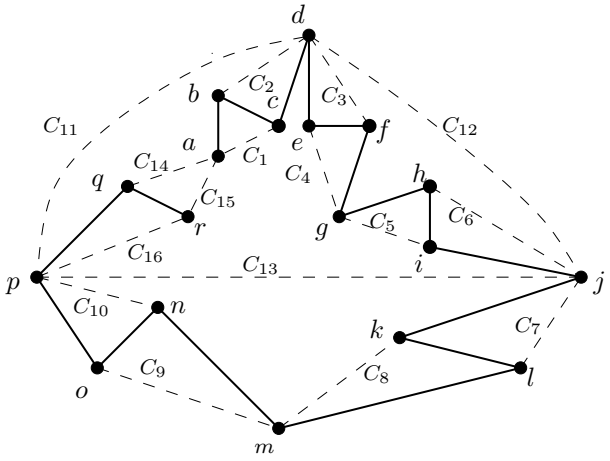


Figure 14: Calculation of DFS tree of G.

$$
\begin{aligned}
C_1 &= \langle a, b, c\rangle \\
C_2 &= \langle b, c, d\rangle \\
C_3 &= \langle d, e, f\rangle \\
C_4 &= \langle e, f, g\rangle \\
C_5 &= \langle g, h, i\rangle \\
C_6 &= \langle h, i, j\rangle \\
C_7 &= \langle j, k, l\rangle \\
C_8 &= \langle k, l, m\rangle \\
C_9 &= \langle m, n, o\rangle \\
C_{10} &= \langle n, o, p\rangle \\
C_{11} &= \langle d, e, f, g, h, i, j, k, l, m, n, o, p\rangle \\
C_{12} &= \langle d, e, f, g, h, i, j\rangle \\
C_{13} &= \langle j, k, l, m, n, o, p\rangle \\
C_{14} &= \langle a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r\rangle \\
C_{15} &= \langle a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q\rangle \\
C_{16} &= \langle c, d, e, f, g, h, i, j, k, l, m, n, o, p\rangle
\end{aligned}
$$

2.2 Computing the Set of Bridges

We choose an arbitrary fundamental circuit from the graph G, for example, $C_{11} = \langle d, e, f, g, h, i, j, k, l, m, n, o, p\rangle$, and compute the set of bridges following the approach in Chapter 4.4.5. See Figure 14.

Let $S \subset G$ be the graph defined by the vertices and edges corresponding to the fundamental circuit C_{11}, that is $S = (\{d, e, f, g, h, i, j, k, l, m, n, o, p\},$

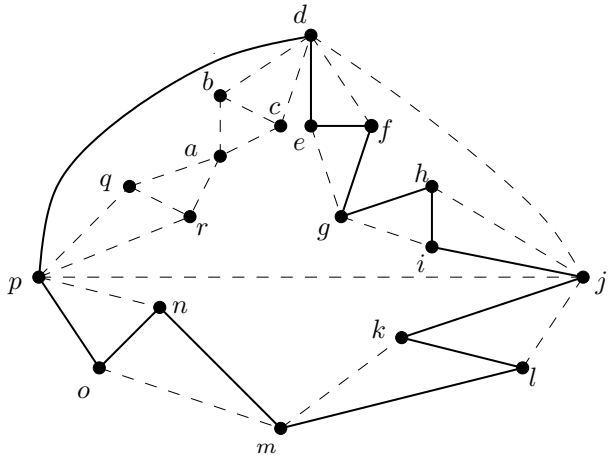


Figure 15: G and fundamental circuit C_{11}.

$\{(d,e),(e,f),(f,g),(g,h),(h,i),(i,j),(j,k),(k,l),(l,m),(m,n),(n,p),(p,d)\}$).

S induces on G a set of components. There is one non-singular component and the rest are singular components. The components of G corresponding to bridges are the following, see Figure 15:

$$
\begin{aligned}
H_1 &= (\{a,b,c,q,r\},\{(a,b),(a,c),(a,q),(a,r)\}) \\
H_2 &= (\{l,j\},\{(l,j)\}) \\
H_3 &= (\{j,h\},\{(j,h)\}) \\
H_4 &= (\{k,m\},\{(k,m)\}) \\
H_5 &= (\{m,o\},\{(m,o)\}) \\
H_6 &= (\{n,p\},\{(n,p)\}) \\
H_7 &= (\{j,p\},\{(j,p)\}) \\
H_8 &= (\{d,j\},\{(d,j)\}) \\
H_9 &= (\{g,i\},\{(g,i)\}) \\
H_{10} &= (\{e,g\},\{(e,g)\}) \\
H_{11} &= (\{d,f\},\{(d,f)\})
\end{aligned}
$$

Thus, H_1, $H_2, \ldots H_{11}$ are also the bridges of G induced by C_{11}.

2.3 Computing the Collapsed Graph

Following the procedure explained in Chapter 4.4.6, bridges are represented by means of a star graph. Figure 16(a) shows the collapsed graph G' corresponding to graph G after replacing every bridge H_i by a star B_i.

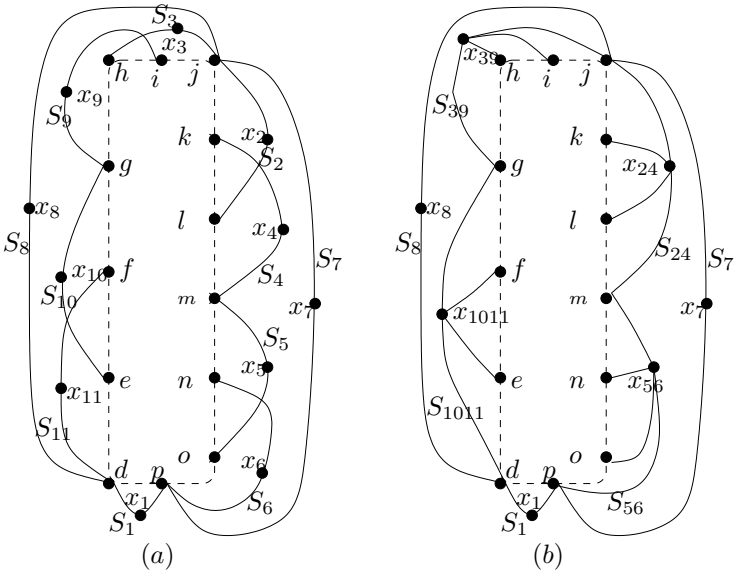


Figure 16: (a) The collapsed graph G' (b) The merged graph G''.

2.4 Computing the Merged Graph

We proceed to merge bridges of G'. Bridges B_2 and B_4 have interlaced attachments and are merged into B_{24}. In a similar way, bridges B_5 and B_6 interlace and are merged into B_{56}, bridges B_3,B_9 interlace and are merged into B_{39} and bridges B_{10},B_{11} are merged into B_{1011}. The resulting merged graph of G', named G'', is shown in Figure 16(b).

2.5 Computing the Planar Embedding of the Merged Graph

Next we compute a planar graph G'''. Figure 17 shows the planar graph G''' as the result of embedding the merged graph G''.

The set of faces defined by the planar embedding of G''' are the following.

$$
\begin{aligned}
f_u &= \{d, x_1, p\} & f_1 &= \{d, x_8, j, x_7, p\} \\
f_2 &= \{d, x_{1011}, g, x_{39}, j\} & f_3 &= \{d, x_{1011}, e\} \\
f_4 &= \{e, x_{1011}, f\} & f_5 &= \{f, x_{1011}, g\} \\
f_6 &= \{g, x_{39}, h\} & f_7 &= \{h, x_{39}, i\} \\
f_8 &= \{i, x_{39}, j\} & f_9 &= \{j, x_7, m, p\} \\
f_{10} &= \{j, x_{24}, k\} & f_{11} &= \{k, x_{24}, l\} \\
f_{12} &= \{l, x_{24}, m\} & f_{13} &= \{m, x_{56}, n\} \\
f_{14} &= \{n, x_{56}, o\} & f_{15} &= \{o, x_{56}, p\}
\end{aligned}
$$

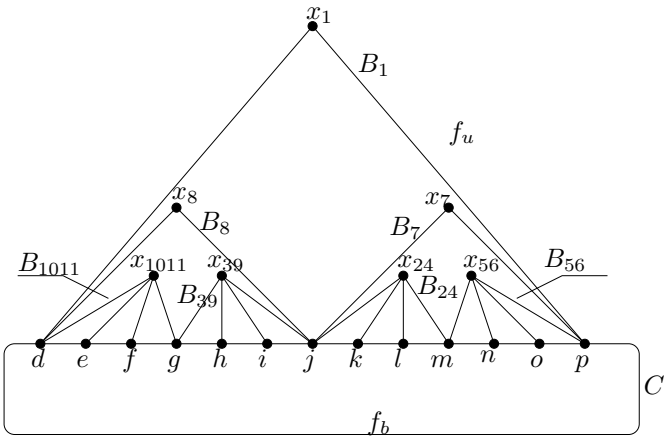


Figure 17: Planar graph G'''.

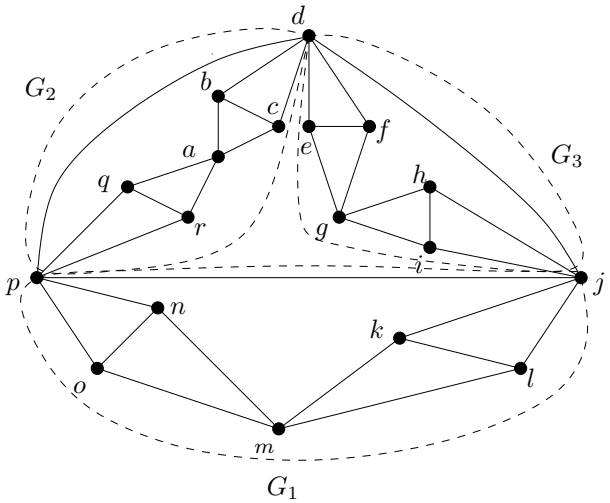


Figure 18: Graph G and clusters G_1, G_2 and G_3.

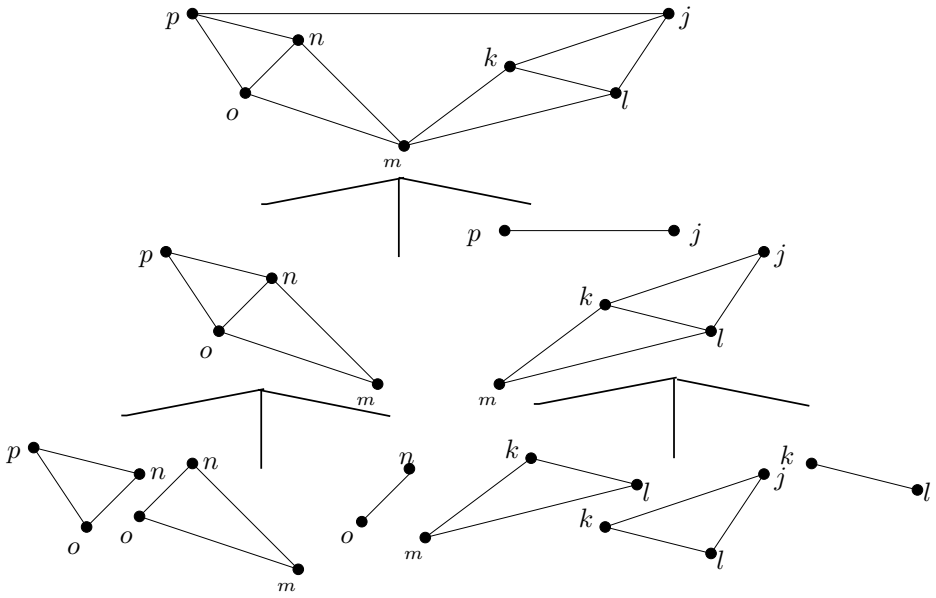


Figure 19: Recursive tree-decomposition of G_1.

2.6 Computing the Hinges

To compute a set of hinges, we search the set of faces $\{f_u, f_1, \ldots, f_{15}\}$. The faces f_1, f_2 and $f_9 \cap f_b$ contains three vertices of the circuit C_{11}. Therefore, the sets $\{d, j, p\}$, $\{d, g, j\}$ and $\{j, m, p\}$ are hinges. Therefore two different decompositions can be performed. If we assume that the algorithm identifies first the triple of vertices $\{d, j, p\}$, the induced graph decomposition into clusters G_1, G_2, G_3 is shown in Figure 18.

2.7 Recursive Tree Decomposition Algorithm

Next the algorithm is recursively applied to graphs G_1, G_2 and G_3. Figure 19 shows a recursive tree-decomposition for G_1 which is similar to G_2, G_3.

3 Case Study Decomposition: A mechanism

In this Section we give an example of application of the DR-planner algorithm to a geometric constraint problem. We use a mechanism, the Watt's linkage.

Figure 20: Watt's linkage in a car suspension (Source: Wikipedia).

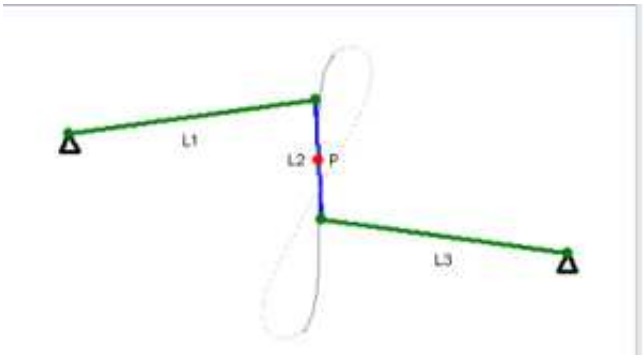


Figure 21: Watt's linkage diagram (Source: Wikipedia).

Watt's linkage is a simple 3-bars linkage that allows to constraint the displacement of a specific point to an approximate straight line. It has been traditionally used in mechanical engineering to force the movement of car suspensions. The Figure 20 shows a picture of an actual car wheels train that uses this linkage.

Watt's linkage can be described by the linkage diagram shown in Figure 21. Point C corresponds to the point on L_1 and L_2 and point D corresponds to the point on L_3 and L_2. Note that the point P displaces vertically.

The geometric constraint system of Figure 21 is structurally well constrained and can be represented by a constraints graph like this of Figure 22. Note that, because the graph is well constrained, it fulfils Laman's Theorem. The set of constraints is:

1. $on(P, L_2)$
2. $on(P_2, L_1)$
3. $on(P_1, L_3)$
4. $d(P_2, C) = d_1$
5. $d(P_1, D) = d_2$
6. $on(C, L_1)$
7. $on(D, L_3)$
8. $\alpha(L_2, L_3)$
9. $\alpha(L_1, L_2)$
10. $d(P, D) = d_3$
11. $d(P, C) = d_3$
12. $on(C, L_2)$
13. $on(D, L_2)$

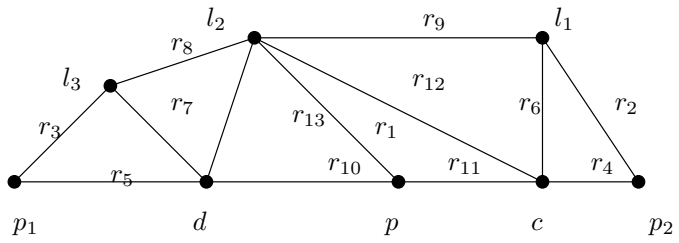
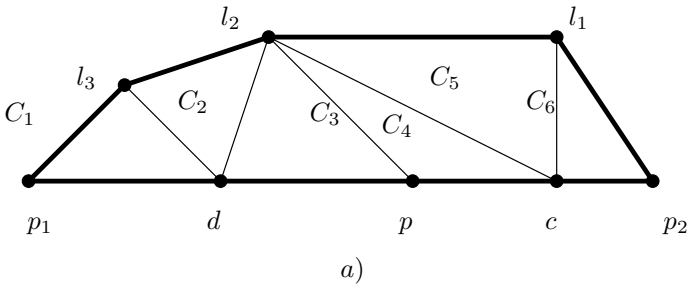
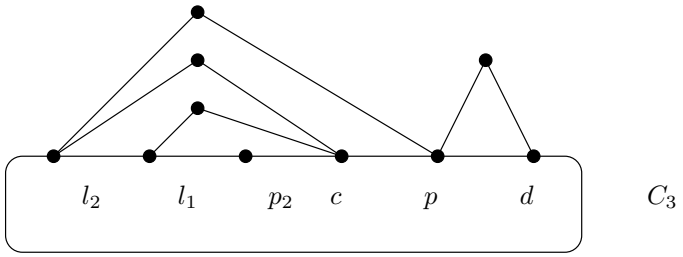


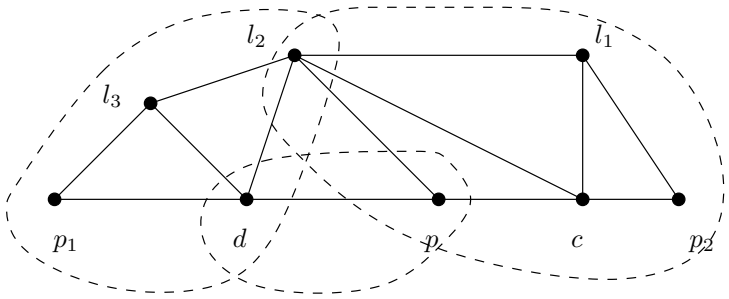
Figure 22: Geometric Constraint graph associated to the Watt's mechanism.



a)



b)



c)

Figure 23: (a) Fundamental Circuits. Circuit C_1 in bold, (b) Collapsed and Merged Graph induced by C_1 (c) One decomposition step.

The set of fundamental circuits induced by the spanning tree in Figure 22 is shown in Figure 23(a). We choose an arbitrary fundamental circuit from the graph G, for example, C_3, and compute the collapsed and merged graph, see Figure 23(b). Figure 23(c) shows one decomposition step.