Theoritical and numerical studies on the graph partitioning problem

Haeder Younis Ghawi Althoby

Normandie Université

THESE

Pour obtenir le diplôme de doctorat

Spécialité mathématiques

Préparée au sein de l'Université de Caen Normandie

Theoretical and Numerical studies on the graph partitioning problem

Présentée et soutenue par
Haeder Younis ALTHOBY

Thèse soutenue publiquement le 6 novembre 2017 devant le jury composé de		
Mme. Fatiha BENDALI-MAILFERT	Maître de Conférences HDR, Université Clermont-Auvergne	Rapporteur
Mme. Sourour ELLOUMI	Professeur, ENSTA-Paristech	Rapporteur
M. Adnan YASSINE	Professeur, Université du Havre	Examinateur
M. Mohamed DIDI BIHA	Professeur, Université de Caen-Normandie	Directeur de thèse

Thèse dirigée par Mohamed DIDI BIHA, laboratoire de Mathématiques Nicolas Oresme

UNIVERSITÉ CAEN NORMANDIE

MISΣ

LMNO

Acknowledgements

I would like to express my deepest gratitude to my supervisor Pr.Mohamed Didi-Biha, for his excellent guidance and extremely generous support. Working with him has been an invaluable experience. He has been very patient and always to answer my questions and teach me the material that I needed to know. When I started writing my thesis, he spared no effort to guide me through a legible and professional mean of putting my thoughts together. He also taught me how the academia works and gave me a lot of helpful advice on my future career. I am very impressed with his limitless enthusiasm and his commitment to always deliver the highest standards for mathematical research. His influence will carry on being present throughout my entire life. Without his help, this thesis would never have come into existence.

I would also like to thank my committee members, Maître de Conférences HDR Fatiha BENDALI-MAILFERT, professor Sourour ELLOUMI, professor Adnan YASSINE for serving as my committee members. I also want to thank you for letting my defense be an enjoyable moment, and for your brilliant comments and suggestions, thanks to you.

Many thanks to all my friends and colleagues in the (Laboratoire de Mathématiques Nicolas Oresme), particularly, George, Frank, André.
A special thanks to my family and my wife's family. Words can not express how grateful. Your prayer for me was what sustained me thus far.

Finally, and most importantly, I would like to thank my wife Massar. Her support, encouragement, quiet patience and unwavering love were undeniably the bedrock upon which the past thirteen years of my life have been built. Her tolerance of my occasional moods is a testament in itself of her unyielding devotion and love.

Contents

List of Figures **iv**

List of Tables **v**

1 Preliminaries and State-of-the-Art **5**

1.1 Combinatorial Optimization 6
1.2 Computational Complextiy 7
1.3 Elements of Polyhedral Theory 9
1.3.1 Branch and Bound Method 14
1.3.2 Cutting Plane Method 14
1.4 Approximation algorithms 17
1.5 Graph Theory 17
1.5.1 The shortest path problem 20
1.5.2 Max-Flow problem 20

2 Graph Partitioning **23**

2.1 Introduction 25
2.2 Formal description of the graph partitioning problem 26
2.3 Hardness 27
2.4 Methods of partition graph 27
2.4.1 Exact methods 28
2.4.2 Heuristic methods 28
2.5 Multilevel Graph Partitioning 33
2.5.1 Coarsening Phase 34
2.5.2 Partitioning Phase 34
2.5.3 Uncoarsening and Refinement Phase 35
2.6 Families of Graph Partitioning Problems 35

2.6.1 The k-way vertex cut problem . . . 35
2.6.2 The k-separator problem . . . 36
2.6.3 The vertex separator problem . . . 36
2.6.4 The multi-terminal vertex separator problem . . . 37
2.6.5 The Multiway Cut problem . . . 37
2.6.6 Vertex multicut problem . . . 38
2.6.7 Critical Nodes Problem (CNDP) . . . 38
2.6.8 The k-way edge cut problem . . . 39
2.6.9 The Balanced Graph Partitioning problem . . . 39

3 Vertex Separator problem 41
3.1 Introduction . . . 42
3.2 Formulation . . . 44
3.3 Neighborhood searching . . . 45
3.3.1 Algorithm 1 . . . 45
3.3.2 Algorithm 2 . . . 46
3.4 Computational experiments . . . 46
3.5 Conclusions and Remarks . . . 53

4 st-Connected vertex separator problem 55
4.1 Introduction . . . 57
4.2 The st-connected separator problem . . . 57
4.3 Formulations . . . 58
4.3.1 Natural Formulation (NF) . . . 58
4.3.2 Extended Formulation (EF) . . . 59
4.3.3 Tree Extended Formulation (TEF) . . . 60
4.4 Polyhedral study . . . 61
4.4.1 Articulation inequalities . . . 65
4.4.2 Partition inequalities . . . 68
4.5 Heuristic . . . 68
4.5.1 Computational experiments . . . 68
4.6 Conclusions and Remarks . . . 74

5 Compte-rendu 77

List of Figures

1.1 The relationship among the classes. 9
1.2 The convex hull of some points in $\mathbb{R}^2$ 11
1.3 Faces and facets. 12
1.4 Cutting Plan Algorithm. 16

2.1 The multilevel approach to graph partitioning. 33
2.2 Different ways to coarsen a graph. 34

3.1 example of VSP 42

4.1 Special articulation assembly configurations v 64
4.2 Special articulation assembly configurations U 66

List of Tables

3.1 Compare our results with 25 instances 49
3.2 DIMACS instances (11 instances) 50
3.3 Comparision of algorithm 2 and BLS algorithms on graphs generated by Helmberg and Rendl (27 instances). 51
3.4 Results of algorithm 2 on graphs generated by Helmberg and Rendl (27 instances) 52

4.1 DIMACS instances (30 instances) 70
4.2 MM-II instances (20 instances) 71
4.3 MM-HD instances (25 instances) 72
4.4 miplib instances (15 instances) 73

Introduction

Combinatorial optimization is a lively field of applied mathematics combining techniques from combinatorics linear programming and the theory of algorithms to solve optimization problems over discrete structures. One of the fundamental concepts of comobinatorial optimization is graph partitioning.

The graph partition problem is defined on data represented in the form of a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. We aim to partition G into smaller components with specific properties. Typically, graph partition problems fall under the category of NP-hard problems. Solutions to these problems are generally derived using heuristics and approximation algorithms.

The graph separator problem is - as the name indicates - the problem of finding good separators in a graph. There are two different types of separators. Sets of edges that separate the graph into k subsets are called edge separators. Sets of vertices that separates the graph into k subsets are called vertex separators. In this thesis all separators are vertex separators unless they are explicitly said to be edge separators.

In this thesis, we consider the *vertex separator problem* (VSP).
Given a graph $G = (V, E)$ be a connected undirected graph and positive integer $\beta(n)$, where $n = |V|$ number of vertices. The vertex separator problem (VSP for short) is to find a partition of V into three nonempty classes A, B, and C such that there is no edge between A and B, $\max\{|A|, |B|\} \leq \beta(n)$, and $|C|$ is minimum. The VSP appears in a wide range of applications such as, Very Large Scale Integration (VLSI) design, telecommunication networks, a separator determines the capacity and the brittleness of the network. Bioinformatics and computational biology, separators in grid graphs provide a simplified representation of proteins.

The problem of finding minimal separators is NP-hard [13, 37].

Several exact algorithms have been proposed for solving VSP ([5, 26, 28, 16]). Most of these algorithms can find optimal results in a reasonable computing time (within 3 hours) for instances with up to 125 vertices, but fail to solve larger instances.
On the other side, many of approximation algorithm introduced to solve the VSP, see ([10, 75, 45, 68, 40]).

In this thesis, we present new algorithms to compute the separators in large instances for undirected graphs ensuring a user defined balance on the induced connected components. Our algorithms use node separators computed by the open source graph partitioning packages. We describe some valid inequalities and derive separation algorithms for these inequalities. Using these results, we develop a Branch-and-Bound algorithm for the problem, along with an extensive computational study which is presented.

We also study the *st*-connected vertex separator problem (*st*-CVS for short). Given a graph $G = (V, E)$ be a connected undirected graph, where $n = |V|$ number of vertices. Let s and t be two disjoint vertices of V, not adjacent. A connected separator of s and t in the graph G is a set $S \subseteq V \setminus \{s, t\}$ such that there are no more paths between s and t in $G[G \setminus S]$, and the graph $G[S]$ is connected.
The *st*-CVS is NP-Hard, and it is in FPT [60]. Narayanaswamy and Sadagopan [63] show that *st*-CVS is NP-Complete on graphs of chordality at least 5 and present a polynomial-time algorithm for *st*-CVS on chordality 4 graphs. They also introduced an algorithm to solve the problem and proved that *st*-CVS parameterized is $W[2]$-hard.

We introduced formulations for the *st*- connected vertex (edge) separator problem. We also find the dimension of the polyhedron for this problem. We presented some valid inequalities. We also presented a heuristic to find minimal *st*-CVS and test it on some libraries of graphs to find the solution of this problem in a short time.

This thesis is organized as follows:
In Chapter 1, we give some basic notions concerning Combinatorial Optimization,

Theory of Complexity, and Polyhedral Theory.

In Chapter 2, we introduce fundamental notions on graph partitioning and then a description of the graph partition and its complextity.
In Chapter 3, we give some basic notions on the vertex separator problem. We then propose two algorithms to solve the vertex separator problem (VSP). These algorithms are done by searching for the neighborhood of any vertex in the graph and choosing the vertex has minimum neighborhood with specific property. We also present the computaional results for three sets of graphs that we take them from the matrix Market library, VSP benchmark instances from the DIMACS challenge on graph coloring, and the set of graphs generated by [43].
In Chapter 4, we give some basic notions concerning the st-connected vertex (edge) separator problem(st-CVS(CES) problem). We then give an overview of this problem on polyhedron. Also dimension and types of valid inequalities on polyhedron are studied. Finally, we provide a heuristic algorithm to solve this problem. They have been conducted in two steps. We first calculate maximum number of disjoint paths between s and t called α_{st}, which is represents the lower bound of any cardinality of any st-separator. We used the code developed by Cherkassky and Goldberg [18] for the maximum flow problem corresponding to calculate α_{st}. We then solved the problem mixed-integer program using Ilog-CPLEX 12.6 [78] to find minimum st-separator. Then check the solution S, if it is connected then is done. Otherwise, we try to find vertices (one or more) that makes S connected.

Chapter 1

Preliminaries and State-of-the-Art

Contents

1.1 Combinatorial Optimization **6**

1.2 Computational Complextiy **7**

1.3 Elements of Polyhedral Theory **9**

1.3.1 Branch and Bound Method 14

1.3.2 Cutting Plane Method 14

1.4 Approximation algorithms **17**

1.5 Graph Theory **17**

1.5.1 The shortest path problem 20

1.5.2 Max-Flow problem 20

In this chapter, we give some basic notions concerning Combinatorial Optimization, Theory of Complexity, and Polyhedral Theory. We then give an overview on two methods that give an exact solution, the BAB and the Cutting Plan methods, and also an overview on approximation methods.

1.1 Combinatorial Optimization

Combinatorial (or discrete) Optimization is one of the most active fields in the interface of Operations Research, Computer Science, and Applied Mathematics.
The target of studying the combinatorial optimization is to find maximize (or minimize) function of many variables subject to constraints. The distinguishing feature of combinatorial optimization is that some of the variables are required to belong to a discrete set, typically a subset of integers.

In general, the problems concerned with combinatorial optimization can be formulated as follows. Let E=$\{e_1, e_2, ..., e_n\}$ be a finite set called *basic set*, where each element e_i is associated with a weight $c(e_i)$. Let Ω be a family of subsets of E. If $\pi \in \Omega$, then $c(\pi) = \sum_{e_i \in \pi} c(e_i)$ denotes the weight of π. The problem is to determine an element of Ω with the larger (or smaller) weights. In the combinatorial optimization problem, the set Ω is called the set of solutions of the problem. In other words,

$$\min(\text{or } \max)\{c(\pi) : \pi \in \Omega\}$$

The term *combinatorial* refers to the discrete structure of Ω. In general, this structure is represented by a graph. The term *optimization* signifies that we are looking for the best element in the set of feasible solutions. In general, this set contains an exponential number of solutions; therefore, we can not expect to solve a combinatorial optimization problem by exhaustively enumerate all its solutions. Such a problem is then considered as a hard problem. The challenge is to develop algorithms that are provably or practically better than enumerating all feasible solutions.

Combinatorial Optimization problems arise in various applications, including Communications Network Design, VLSI design, Machine Vision, Airline Crew Scheduling, Corporate Planning, Computer-Aided Design and Manufacturing, Database Query Design, Cellular Telephone Frequency Assignment, Constraint Directed Reasoning, and Computational Biology. Besides the applications, discrete optimization has aspects that connect it with other areas of mathematics (e.g., Algebra, Analysis and Continuous Optimization, Geometry, Logic, Numerical Analysis, Topology, and, of course, other subdisciplines of Discrete Mathematics such as Linear and Integer

Programming, Graph Theory, Artificial Intelligence, and Number Theory). All these problems, when formulated mathematically as the minimization or maximization of a certain function defined on some domain, have a commonality of discreteness.

In fact, combinatorial optimization is closely related to Algorithm Theory and Computational Complexity Theory as well. The next section introduces computational issues of combinatorial optimization.

1.2 Computational Complextiy

Computational Complexity Theory is a branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying computational problems according to their inherent difficulty and relating those classes to each other. This theory was early born by Gabriel Lame in 1844, but the field really began to flourish in 1962 by Edmonds[30] and Cook in 1971[19] that they offered a framework to classify problems according to their difficulty. In 1972, Karp[48] introduced more information about 21 diverse combinatorial and graph theoretical problems, each infamous for its computational intractability, which are NP-complete in the Complexity Theory.

A *problem* is a general question to which we wish to find an answer. This question usually has parameters or variables, the values of which have yet to be determined. A problem is posed by giving a list of these parameters as well as the properties to which the answer must conform.
An *instance* of a problem is obtained by giving explicit values to each of the parameters of the instanced problem.
An *algorithm* is a sequence of elementary operations that, when given an instance of a problem as input, give the solution of this problem as output after execution of the final operation.

A *decision problem* is question concerning the existence, for a given instance, of a configuration such that this configuration itself or its value conforms to certain properties. The solution to a decision problem is an answer to the question associated with the problem. In other words, this solution can be *yes*, such a solution does

exist or *no*, such a solution does not exist.
An *algorithm* is any well-defined computational procedure that takes some values, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output. The number of input necessary parameters to describe an instance of a problem called *size* of problem .

The efficiency of an algorithm, evaluated as the amount of computing resources it requires, decides whether an algorithm is good or not. In the efficiency of an algorithm, there are two principal resources : (a) time (i.e. number of elementary computational steps (how much time the best algorithm requires to solve the problem)); (b) space (i.e. magnitude of internal memory required).

The computational complexity of a problem can be related to the algorithm that solves it. Moreover, any problem can be classified into some complexity classes according to its complexity. The most well-known complexity classes **P-class** (*polynomial*) and **NP-class** (*None-deterministic Polynomial*) are collections of decision problems. The class **P** is a complexity class represents the set of all decision problems that can be solved in polynomial time. That is, given an instance of the problem, the answer "yes" or "no" can be decided in polynomial time.
The class **NP** is a complexity class represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time. This means that if an instance of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time. It can be seen that the class **P** belonges to **NP**. The open question is whether or not **NP** problems have deterministic polynomial time solutions.
The class $\mathbf{NP}-Complete$ is a complexity class represents the set of all problems X in **NP** for which it is possible to reduce any other **NP** problem Y to X in polynomial time.
Intuitively, this means that we can solve Y quickly if we know how to solve X quickly. Precisely, Y is reducible to X, if there is a polynomial time f to transform instances y of Y to instances $x = f(y)$ of X in polynomial time with the property that the answer to y is *yes* if and only if the answer to $f(y)$ is yes.
The class **NP-hard** is that a problem X is **NP-hard**, if there is an $\mathbf{NP}-Complete$ problem Y, such that Y is reducible to X in polynomial time. But since any NP-

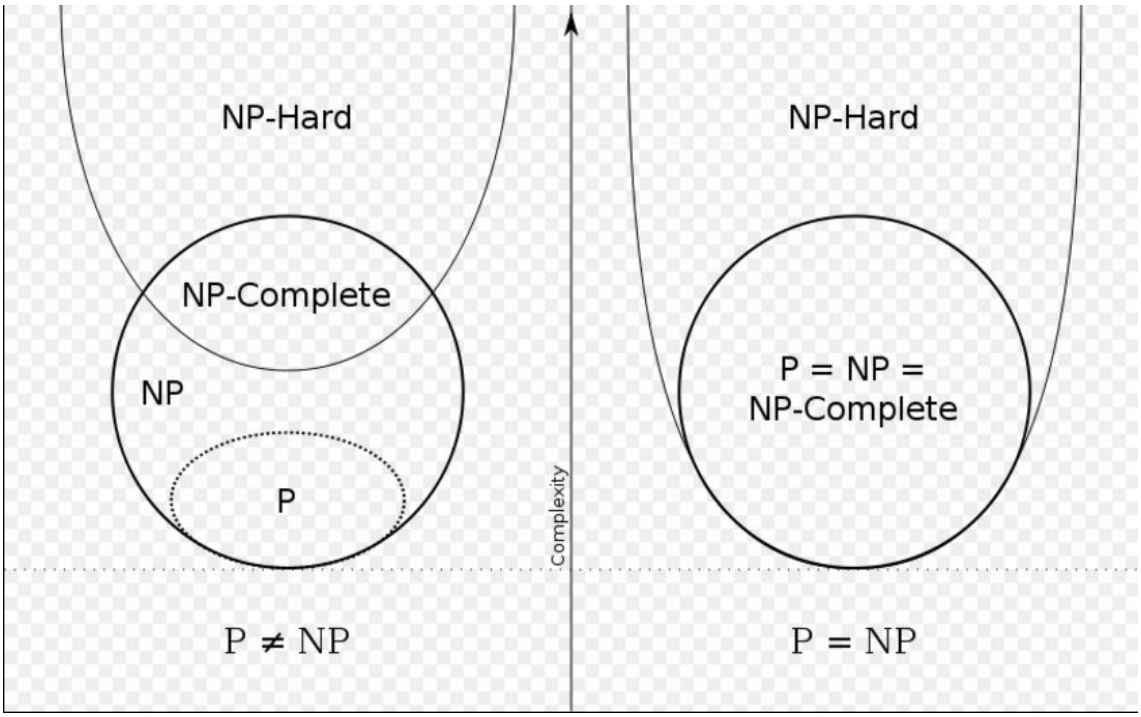


Figure 1.1 – The relationship among the classes.

Complete problem can be reduced to any other NP-Complete problem in polynomial time, all NP-Complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.
Figure 1.1 shows the relationship among these classes.

We note that most of combinatorial optimization problems are **NP-hard**. One of the most efficient approaches developed to solve those problems is the so-called Polyhedral Approach.

1.3 Elements of Polyhedral Theory

In this section, we give some fundamental notions of polyhedral theory on linear and integer optimization. Further references are Schrijver [69], Nemhauser and Wolsey [64], and Mahjoub [59].

Polyhedral Theory deals with the feasible sets of Linear Programming Problems, which are called *polyhedra*. Now, Polyhedral Theory may be viewed as a part of convex analysis, which is the branch of mathematics that studies convex sets, (i.e., sets that contain the line segments between each pair of its points). There are two ways of defining polyhedra and polytopes. Some texts begin with the definition of a polytope as a convex combination of a finite number of points. Others, begin with the definition of polyhera via the intersection of finitely many half spaces. Polyhedra appeared early in the history (such as Pyramids, the dices, and other signs of civilization), and in geometry (as Plato, Euclid, Archimedes). However, all these were individual, and particular polyhedra. Even in the middle of the 18^{th} century, Euler discussed his famous theorem *(V-E+F =2)* without specifying what are the polyhedra to which it applies. During the 19^{th} century, the center of attention concerning polyhedra switched to (more-or-less explicitly declared) convex ones.

We shall first recall some definitions and properties related to polyhedral theory.

Given n be a positive integer and $x \in \mathbb{R}^n$. We say that x is a *linar combination* of $x_1, x_2, ..., x_n \in \mathbb{R}^n$ if there exist m scaler $\delta_1, \delta_2, ..., \delta_m$ such that $x = \sum_{i\in 1}^{m} \delta_i x_i$. If $\sum_{i\in 1}^{m} \delta_i = 1$, then x is said to be an *affine combination* of $x_1, x_2, ..., x_m$. if $\delta_i \geq 0$, for all $i \in \{1, ..., m\}$, we say that x is a *conic combination* of $x_1, x_2, ..., x_m$. if x is *affine* and *conic* ,we say that the x is *convex combination.*
Given a set $S = \{x_1, x_2, ..., x_m\} \in \mathbb{R}^{n\times m}$, the *convex hull* of S is the set of points $x \in \mathbb{R}^n$ which are convex combination of $x_1, x_2, ..., x_m$, that is
$conv(S) = \{x \in \mathbb{R}^n \mid x \text{ is a convex combination of } x_1, x_2, ..., x_m\}$ (see Figure 1.2).

The points $x_1, x_2, ..., x_m \in \mathbb{R}^n$ are *linearly independents* if the unique solution of the system $\sum_{i=1}^{m} \delta_i x_i = 0$ is $\delta_i = 0$, for all $i \in \{1, ..., m\}$. These points are *affinely independent* if the unique solution of the system

$$\begin{aligned} \textstyle\sum_{i=1}^{m} \delta_i x_i &= 0, \\ \textstyle\sum_{i\in 1}^{m} \delta_i &= 1, \end{aligned}$$

is $\delta_i = 0$, where $i = 1, ..., m$.

A subset P of $\mathbb{R}^n$ is called a *polyhedron* if there exists an $m \times n$ matrix A and a vector $b \in \mathbb{R}^n$ (for some $m \geq 0$) such that $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$. Thus, P is a

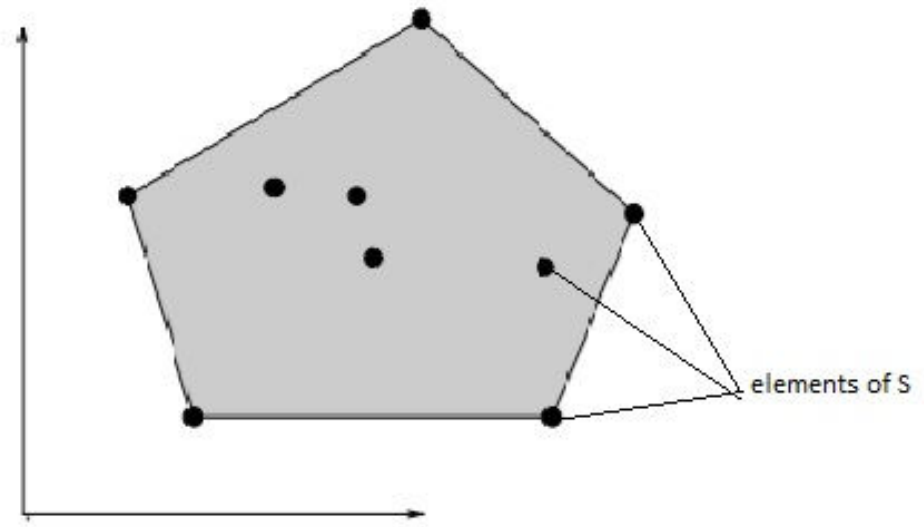


Figure 1.2 – The convex hull of some points in $\mathbb{R}^2$

polyhedron if and only if P is the set of solutions of the linear system $Ax \leq b$. A point x of P will be also called a *solution* of P. Any inequality $\alpha^T x \leq \beta$ is called *valid* for P if $\alpha^T x \leq \beta$ holds for each $x \in P$. A polyhedron P is *bounded* if there exist $l, u \in \mathbb{R}^n$ such that $l \leq x \leq u$ for any $x \in P$. A subset P of $\mathbb{R}^n$ is called a *polytope* if P is the convex hull of finitely many vectors in $\mathbb{R}^n$. Moreover a set P is a polytope if and only if P is a bounded polyhedron.
A polyhedron P of $\mathbb{R}^n$ is said to be *k-dimensional* if the maximum number of affinely independent points of P is $k+1$. We then write $dim(P) = k$. If dim(P)=n, then the polyherdron P is said to be *full-dimensional*.
A linear inequality $ax \leq \alpha$ is said to be *valid* for a polyhedron $P \in \mathbb{R}^n$ if it is satisfied by every point of P, that is $P \subseteq \{x \in \mathbb{R}^n : ax \leq \alpha\}$.
Let $ax \leq \alpha$ is a valid inequality, then the polyhedron:

$$F = \{x \in P : ax = \alpha\}$$

is called a *face* of P, and we say that F is defined by the inequality $ax \leq \alpha$. The empty set and the polytope P itself are considered as faces of P. A face of P is said

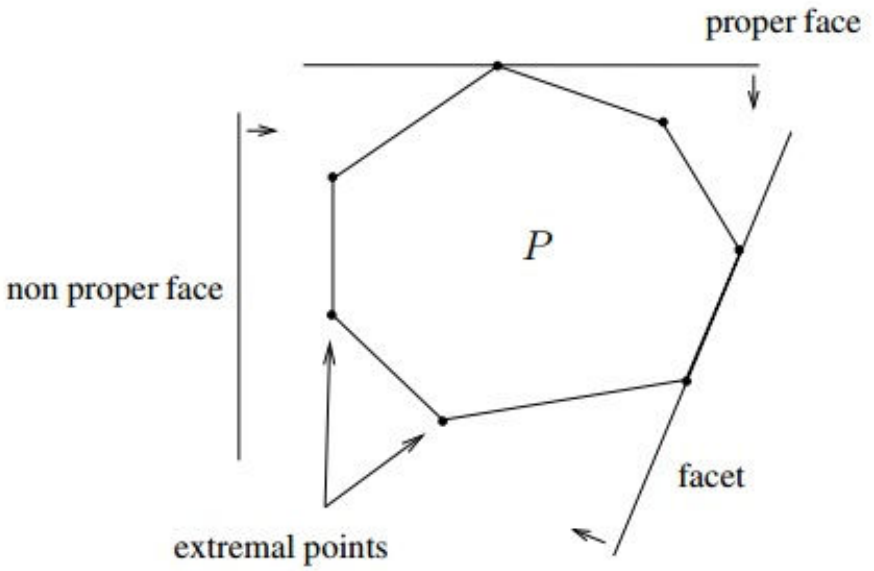


Figure 1.3 – Faces and facets.

to be *proper* if it is non-empty ($F \neq \phi$) and different from P ($F \neq P$)(see Figure 1.3).

If F is a proper face and $dim(F) = dim(P) - 1$, then F is called a *facet* of P. We also say that $ax \leq \alpha$ induces a facet of P or is a *facet definin* inequality.

For each facet F of P, one of the inequalities representing F is necessary in the description of P. A vector $x \in \mathbb{R}^n$ is an *extreme point* of a pointed polyhedron P if it cannot be written as a convex combination of the other vectors in P. In other words, a point x of a polyhedron P is said to be an *extreme point* if there are not two solutions x^1 and x^2 of P,$(x^1 \neq x^2)$, such that $x = \frac{1}{2}x^1 + \frac{1}{2}x^2$. It is equivalent to say that x induces a face of dimension 0. A polyhedron P has a finite number of extreme points.
We say that the vector $x \in \mathbb{R}^n$ that satisfies $Ax \leq b$ is a *feasible solution* for the linear system $Ax \leq b$. The feasibility of a system $Ax \leq b$ of linear inequalities is characterized by Farkas′ lemma.

Theorem 1.3.1 *(Farkas′ Lemma). The linear system $Ax \leq b$ with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^n$ is a feasible if and only if $(y^T b \geq 0)$ holds for any $(y \geq 0)$ with $(y^T b = 0)$.*

Linear optimization problem concerns maximizing or minimizing a linear function $c^T x$ over a polyhedron P has the form

$$\max\{c^T x : x \in P\}.$$

The polyhedron P is called the feasible region, and any vector in P is a feasible solution. If the feasible region is nonempty, the problem is called *feasible* and *infeasible* otherwise. The linear function $c^T x$ is called *objective function.*
If it is known that $P = \{x \in \mathbb{R}^n : Ax \leq b\}$, the optimization problem also has an equivalent form as below.

$$max\{c^T x : Ax \leq b, x \geq 0\}. \tag{1.3.1}$$

A vector $x \in \mathbb{R}^n$ is called *integer* if each component is an integer, i.e., if x belongs to $\mathbb{Z}^n$. Many combinatorial optimization problems can be described as maximizing a linear function $c^T x$ over the integer vectors in some polyhedron $P = \{x | Ax \leq b\}$. Therefore, this type of problems can be described as:

$$max\{c^T x : Ax \leq b, x \in \mathbb{Z}^n\}. \tag{1.3.2}$$

are called *integer linear programming.* They consist of maximizing a linear function over the intersection $P \cap \mathbb{Z}^n$ of a polyhedron P with the set $\mathbb{Z}^n$ of integer vectors.

No polynomial-time algorithm is known to exist for solving an integer linear programming problem in general. In fact, the general integer linear programming problem is NP-complete.
A polyhedron P is called an *integer polyhedron* if it is the convex hull of the integer vectors contained in P. This is equivalent to P is rational and each face of P contains an integer vector. So a polytope P is integer if and only if each vertex of P is integer. If a polyhedron $P = \{x : Ax \leq b\}$ is integer, then the linear programming problem $max\{c^T x : Ax \leq b\}$ has an integer optimum solution if it is finite. Hence, in that case,

$$max\{c^T x : Ax \leq b, x \in \mathbb{Z}^n\} = max\{c^T x : Ax \leq b\}.$$

1.3.1 Branch and Bound Method

The *Branch and bound algorithm* provides an appropriate approach that is for virtually all combinatorial problems. The Branch and Bound algorithms are almost always *primal* in the sense that they proceed from one feasible solution to another until optimality is verified. In fact, they often find optimal or near optimal solutions early in the enumeration process and spend the majority of the time verifying optimality. The Branch and Bound has four fundamental elements. These elements are:(a) separation (partition) into subproblems, (b) relaxation (upper bounding), (c) fathoming of subproblems (lower bounding), and (d) selection of subproblems (branching).
To describe these elements, we need to define the set of the feasible solutions by

$$S = \{x : c^T x : Ax \leq b, x \in \mathbb{Z}^n\}.$$

Instead of attempting to solve the problem directly over S, the set is successively divided into smaller and smaller sets which have the property that any optimal solution must be in at least one of the sets. This process is called separation. For each subproblem we create, we compute its subproblems as the upper bound. If we solve the LP relaxation, we obtain a solution x^0, which in general is not integer. the cost $c(x^0)$ of this solution x^0 is a lower bound on the optimal cost $c(x^*)$ (where x^* is the optimal solution to problem 0), and if x^0 was integer, we would be done.

1.3.2 Cutting Plane Method

One of the general methods for solving integer linear programming problems called a *cutting plane algorithm*. The idea of this algorithm is to add cutting planes to a linear program, (i.e., valid inequalities), in order to approximate the underlying integer program.

Historically, an early version of this class of algorithms was the Gomory Cutting Plane Algorithm from around 1960. The new approach to cutting plane algorithms consists of finding theoretically a class of strong valid inequalities for an integral polytope before the computations starts, and then constructing separation algorithms

to test whether some current solution violates one of these inequalities. Thus, the principle is to combine problem specific inequalities with the general framework of separation and optimization.

In combinatorial optimization problem, it is generally difficult to characterize the associated polyhedron by a system of linear inequalities. Moreover, if the problem is NP-complete, there is very little hope of obtaining such a description. Furthermore, even if it is characterized, the system describing the polyhedron can contain a very large (even exponential) number of inequalities. Therefore, it cannot be totally used to solve the problem as a linear program. However, by using a cutting-plane method, a partial description of the polyhedron can be sufficient to solve the problem optimally.

Consider an Integer Linear Program (ILP) of the form (1.3.2), associated with the ILP are two polyhedras:

- $P := \{x \in \mathbb{R}^n : Ax \leq b\}$ (the feasible region of the linear programming relaxation), and
- $P_I := \{x \in \mathbb{Z}^n : Ax \leq b\}$ (the convex hull of feasible integer solutions).

A *cutting plane* is a linear inequality which is satisfied by all points in P_I, but it is not satisfied by all points in P. A primal cutting plane algorithm begins with a feasible solution to the ILP denote by $\hat{x}$. The $\hat{x}$ must be an extreme point of both P and P_I. If no such $\hat{x}$ is known in advance, then artificial variables must be used to find one. If $\hat{x}$ is a dual feasible, $\hat{x}$ is an optimal and the method stops. If not, then a non-basic variable with negative reduced cost is selected to come into the basis. Also the usual primal simplex criterion is used to select a basic variable to leave the basis. Then a primal simplex pivot is made, leading to a new vector x^*. If x^* is integral, then it represents an improved ILP solution, and it becomes the new $\hat{x}$. If on the other hand it is fractional, then a cutting plane is generated which cuts x^*. Then, another attempt is made to pivot from $\hat{x}$, leading to a different x^*, and so on. The method terminates when $\hat{x}$ has been proved to be dual feasible. Figure(1.4) show that Cutting Plan Algorithm.

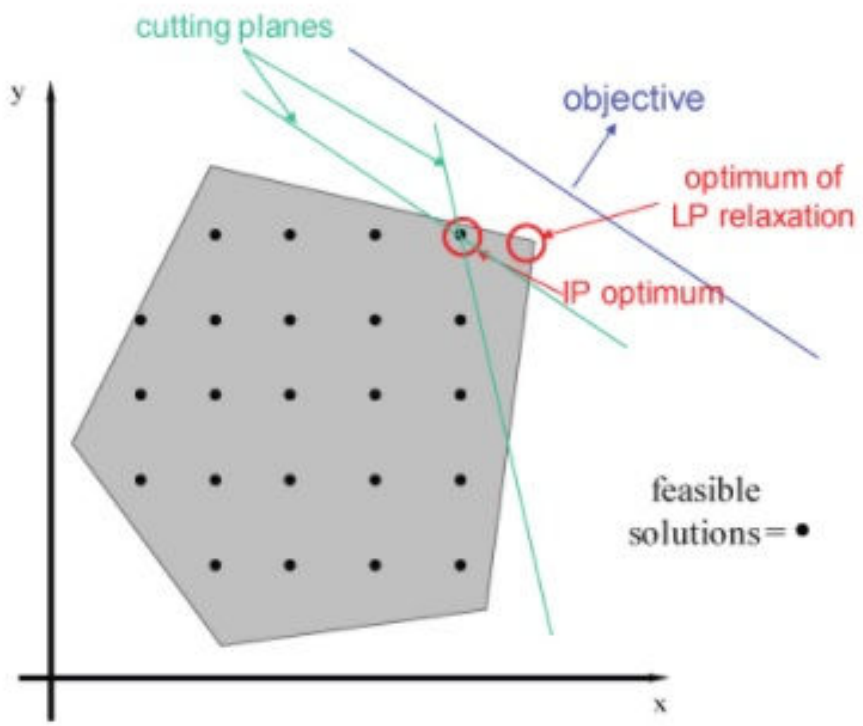


Figure 1.4 – Cutting Plan Algorithm.

1.4 Approximation algorithms

For NP-hard integer optimization problems, finding an optimal solution in general is not possible, and it may still be possible to find *near-optimal* solutions in polynomial time. In practice, near-optimality is often good enough. An algorithm that returns near-optimal solutions is called an *approximation* algorithm. However, these methods do not come with rigorous guarantees concerning the quality of the final solution or the required maximum runtime. The design of good approximation algorithms is a very active area of research to find new methods and techniques. It is quite likely that these techniques will become of increasing importance in tackling large real-world optimization problems. The quality of an approximation algorithm is the maximum distance between its solutions and the optimal solutions evaluated over all the possible instances of the problem.

Heuristic local search methods, such as tabu search and simulated annealing or greedy algorithm are often quite effective at finding near-optimal solutions.
However, these methods do not come with rigorous guarantees concerning the quality of the final solution or the required maximum runtime. Approximation algorithms generally provide a feasible solution to a problem instance in polynomial time.

An *approximation scheme* for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1+\epsilon) - approximation$ algorithm. We say that an approximation scheme is a *polynomial-time approximation scheme* if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

1.5 Graph Theory

The problems that have been studied the most in the combinatorial optimization involve looking for certain structures in graphs. Furthermore, many applied problems in computer science, VLSI, telecommunications, scheduling, etc. are fruitfully modeled by graphs, possibly through some kinds of optimization problem in that graph.

In this section, we give a brief introduction to some basic concepts and results in

graph theory, and two of the fundamental classical optimization problems in graphs (the Flow Max problem and the Shortest Path problem) with focus on the mathematical ideas underlying efficient algorithms for solving this problem. These concepts will be used throughout on the next chapters. Books that can be recommended for further reading include [6] and [17].

A graph is an ordered pair $G = (V, E)$, where V is a finite set with elements called *nodes* or *vertices* and E is a finite set of unordered pairs from V, each such pair is called an *edge* or an *arc*. Any graph may be visualized in $\mathbb{R}^2$ by drawing the nodes as distinct points and each edge as a curve joining two nodes. Such a drawing of the graph is called an *embedding* of the graph. The number n of nodes is called the *order* of the graph, and the number m of edges is called the *size* of the graph. For an edge $e = \{u, v\} \in E$, we write $e = uv$. Since e is an unordered pair, we have $uv = vu$. When $e = uv$ we say that e is between (or joins) u and v, and that u and v are incident to e. Two nodes are *adjacent* (or are neighbors) if there is some edge between them, and these nodes are called the *end nodes* of that edge. Two edges are neighbors if they share a node.
A node v and an edge e are incident if v is an end node of e.

Consider a graph $G = (V, E)$ and define the node-edge incidence matrix $A \in \mathbb{R}^{n \times m}$ with elements being 0 or 1 as follows. A has one row for each node i and a column for each edge e, and $a_{i,e} = 1$ if i is incident to e (i.e., i is an end node of e) and $a_{i,e} = 0$ otherwise. All the ones in row i of A are in columns that correspond to the set $\delta(i)$ of edges being incident to node i, this set is often called the *star* of node i. The row sum $d(i) = d_i = \mid \delta(i) \mid$ is the number of edges incident to i(= the number of neighbor nodes to i), we call this number the *degree* of node i. More generally, for $S \subset V$, we let $\delta(S)$ denotes the set of edges with one end node in S and the other end nodes in $S^* = V \setminus S$, such a set is called a *cut*.
An *isolated* node is a node of degree 0, (i.e., without neighbors). A walk (in a graph) is a node-edge sequence $W : v_0, e_0, v_1, e_1, ..., v_{n-1}, e_{n-1}, v_n, e_n, v_{n+1}$ where $e_i = \{v_i, v_{i+1}\} \in E$ for $i = 0, ..., n$. The end nodes of W are v_0 and v_n, and we call W is a $v_0v_n - walk$ or a *walk* between v_0 and v_n. In a walk, we may have repeated nodes or edges. If, however, there are no repeated nodes, we call the walk is a *path* P between v_0 and v_n (or a $v_0v_n - path$). All the nodes in a path, except the two end nodes, are called *internal nodes*. The *length* of a walk (path) is its number of edges.

There is another matrix which may be used to represent the graph, which is used in our thesis. Let the graph $G = (V, E)$ have node set$\{v_1, ..., v_n\}$. The *adjacency matrix* $A \in \mathbb{R}^{n \times n}$ is a $0/1 - matrix$ where the $(i, j)'$th element is 1 if $\{v_i, v_j\} \in E$ and 0 otherwise. We note that A is symmetric and that it has zeros on the diagonal. The powers of this matrix contains information about walks in the graph.

Consider a graph $G = (V, E)$ and let $U \subseteq V$ and $F \subseteq E$. The subgraph induced by U, denoted $G(U)$, is the graph $(U, E(U))$, where $E(U)$ consists of those edges in E with both end nodes in U. We then call $(U, E(U))$ a *node-induced subgraph.* If $F \subseteq E$ the graph $G(F) = (V, F)$ is called an e*dge-induced subgraph.* The *complement* of a graph $G = (V, E)$ is the graph $G^* = (V, E^*)$, where $E^* = \{uv : uv \notin E\}$.
A basic graph property is connectivity. We say that two nodes u and v in a graph G are *connected* if G contains an $uv - path$. This gives rise to an equivalence relation on V. We write uCv if u and v are connected, and then this binary relation is an equivalence relation so

(i) uCu,

(ii) $uCv \Rightarrow vCu$,

(iii) $(uCv \text{ and } vCw) \Rightarrow uCw$.

This equivalence relation gives rise to a partition of V into subsets $V_1, ..., V_p$ being the maximal connected subsets of V. These sets are called the *(connected) components* of G. Thus, two nodes are connected if and only if they are in the same component. Let $c(G)$ denote the number of components of G. A graph is called *connected* if $c(G) = 1$, (i.e., if each pair of nodes is connected). We note that each isolated node is also a component.
The *complete graph* on n nodes, denoted K_n, is the graph with n nodes where each pair of nodes are adjacent. Thus K_n has $n(n-1)/2$ edges. A *bipartite* graph is a graph where the nodes may be divided into two sets V_1 and V_2 such that each edge has one end node in V_1 and the other end node in V_2. More generally, the *k-partite graph* $K(n_1, ..., n_k)$ is the graph with node set consisting of (disjoint) sets, or color classes, V_i with $\mid Vi \mid = n_i$ for $i = 1, ..., k$, and where the end nodes of each edge belong to different color classes. Thus, a 2-partite graph is the same as a bipartite graph. A k-partite graph is complete if each pair of nodes lying in different color

classes are adjacent.
A *directed graph* (or *digraph*), is an ordered pair $D = (V, A)$ where V is a finite set of nodes and A is a finite set of ordered pairs of nodes. Each such ordered pair $e = (u, v)$ is called an *arc* and u is called the *initial end node*, (or *tail*), of e and v is called the *terminal end node*, (or *head*), of edge e.

1.5.1 The shortest path problem

Let $G = (V, E)$ be a graph. The length of any path in G is the number of its edges. For $s, t \in V$, the distance from s to t is the minimum length of any $s - t$ path. The minimum length of an $s - t$ path is equal to the maximum number of disjoint $s - t$ cuts.
Let $e_{i,j}$ be the edge incident to both v_i and v_j. Given a real valued weight function $f : E \to \mathbb{R}$, and an undirected graph G, the shortest path from s to t is the path $P = (v_1, v_2, \ldots, v_n)$ (where $v_1 = s$ and $v_n = t$) that over all possible n minimizes the sum $\sum_{i=1}^{n-1} f(e_{i,i+1})$. When each edge in the graph has unit weight or $f : E \to \{1\}$, this is equivalent to finding the path with the fewest edges.

Many efficient algorithms are used to find this problem like *Dijkstra's algorithm*. Which solves the single-source shortest path problem. The*Bellman-Ford algorithm* solves the single-source problem if edge weights may be negative. The A^* *search algorithm* solves for single pair shortest path using heuristics to try to speed up the search. The *Floyd-Warshall* algorithm solves all the pairs of the shortest paths. The *Johnson's algorithm* solves all pairs shortest paths. The *Viterbi algorithm* solves the shortest stochastic path problem with an additional probabilistic weight on each node.
There are many application of this method, for more details see [21] .

1.5.2 Max-Flow problem

The maximum flow problem was first formulated in 1954 by T.E. Harris and F. S. Ross [42] as a simplified model of Soviet railway traffic flow. In 1955, Lester R. Ford, and Delbert R. Fulkerson created the first known algorithm, the *Ford-Fulkerson al-*

gorithm [35]. Over the years, various improved solutions to the maximum flow problem were discovered, notably the shortest augmenting path algorithm of Edmonds and Karp and independently Dinitz; the blocking flow algorithm of Dinitz; the push-relabel algorithm of Goldberg and Tarjan; and the binary blocking flow algorithm of Goldberg and Rao. The electrical flow algorithm of Christiano, Kelner, Madry, and Spielman finds an approximately optimal maximum flow but only works in undirected graphs.

Consider a graph $G = (V, E)$ with a cost function $\omega : V \times V \longrightarrow \mathbb{R}$ denoting capacities on the edges. The nodes $s \in V$ and $t \in V$ are two designated notes in G and are also called *source* and *sink*. A *flow* is a function $f : E \longrightarrow \mathbb{R}^+$ which satisfies the *capacity constraint*. The capacity constraint demands that for each edge $(u, v) \in E$ the inequation $0 \leq f(u, v) \leq \omega(u, v)$ holds. The value $val(f)$ of a flow f is defined as $val(f) := \sum_{(s,v)\in E} f(s, v) - \sum_{(v,s)\in E} f(v, s)$ which is equal to the amount of flow routed from source to sink. The max-flow problem demands to transfer as much flow as possible from source s to sink t. This means we want to maximize the value $val(f)$ of a flow f in the network.
In fact,there is a relation between the max-flow and minimum cut edges (or vertices) in a parition graph, one of the important theorems is called **Menger's theorem**.

Theorem 1.5.1 ***(Menger's theorem)****. The maximum number of arc-disjoint st-dipaths in a directed graph equals the minimum number of arcs in an st-cut.*

There are many of real world application of this method, for more detailes see [69].

Chapter 2

Graph Partitioning

Contents

2.1 Introduction **25**

2.2 Formal description of the graph partitioning problem **26**

2.3 Hardness **27**

2.4 Methods of partition graph **27**

2.4.1 Exact methods 28

2.4.2 Heuristic methods 28

2.5 Multilevel Graph Partitioning **33**

2.5.1 Coarsening Phase 34

2.5.2 Partitioning Phase 34

2.5.3 Uncoarsening and Refinement Phase 35

2.6 Families of Graph Partitioning Problems **35**

2.6.1 The k-way vertex cut problem 35

2.6.2 The k-separator problem 36

2.6.3 The vertex separator problem 36

2.6.4 The multi-terminal vertex separator problem 37

2.6.5 The Multiway Cut problem 37

2.6.6 Vertex multicut problem 38

2.6.7 Critical Nodes Problem (CNDP) 38
2.6.8 The k-way edge cut problem 39
2.6.9 The Balanced Graph Partitioning problem 39

In this chapter, we introduce fundamental notions on graph partitioning and then the description of the graph partition and its complextity of it. We then perform a brief summary of method partition that is exact and heuristic. We also give stages of Multilevel graph partitioning in details. We finish this chapter with families of graph partitioning problem and introduce a brief summary of each one of it.

2.1 Introduction

Graphs are structures formed by a set of vertices (also called *nodes*) and a set of edges that are connections between pairs of vertices, while the word *partitioning* represents the action of creating a partition by segmenting a set into several parts. In mathematics, a partition of a set X is a family of pairwise disjoint parts of X whose union is the set X. Therefore, each element of X belongs to one, and only one of the parts of the partition.
Creating a partition consists of distributing a set of objects into several subsets. In order to distribute the objects in these different subsets, it may be useful to compare them.

Graph partitioning is a discipline situated between computer science and applied mathematics, and *the graph partition* problem is defined on data represented in the form of a graph $G = (V, E)$, with V vertices and E edges, such that it is possible to partition G into smaller components with specific properties.
An important implication of the separator theorem is that any graph with a fixed excluded minor with maximum degree bounded by d can be partitioned into small components of size at most $\text{poly}(d, 1/\varepsilon)$ by removing only an ε-fraction of edges.

There are many applications of graph partitioning in different fields, likes software and hardware design (e.g. Parallel Processing [12] and Physical design of digital circuits for Very Large-Scale Integration (VLSI) [50, 20], Computer Aided Design(CAD) [20]), networks and road design [52, 51](For example, edges could be road segments and nodes intersections.), Image Processing [67, 15] (partition the pixels of an image into groups that correspond to objects.), biology[46, 62] (geographic information services and physical mapping of DNA (Deoxyribo Nucleic Acid, basic constituent of the gene)), and more. For example,in VLSI design, it is often required to split designs too large to fit in a single chip into many smaller parts of "roughly" equal size, where each part is mapped onto a separate chip. Compared to connections within a chip, connections between chips are costly, slow, and power consuming. Hence, it is desirable to reduce the capacity of the cut defined by a partition.
In parallel computation applications, a computation is modeled by a graph (i.e. dataflow graph), a vertex models a computation of an expression and an edge models

communication of a result of one expression to an operand of another expression. Partitioning determines the mapping of computations to processors. As opposed to cut edges, edges between vertices mapped to the same processor do not require communication between processors. Since communication between processors is slow, it is again desirable to reduce the cut.

2.2 Formal description of the graph partitioning problem

As mentioned in the previous section, a graph is a pair formed by a set of vertices and a set of edges. It is therefore possible to partition, in a mathematical sense, the set of vertices as well as the set of edges. However, although some problems seek to partition the edges of a graph, graph partitioning is mostly understood as the partition of the vertices of the graph.
Let $G = (V, E)$ be a graph with node weights $f : V \longrightarrow \mathbb{R}$, $n =\mid V \mid$ and $m =\mid E \mid$ and $P_k = \{V_1, ..., V_k\}$ a set of k subsets of V. P_k is said to be a partition of G if :

- no element of P_k is empty :
  $\forall i \in \{1, ..., k\} \neq \phi$.
- the elements of P_k are pairwise disjoint :
  $\forall (i, j) \in \{1, ..., k\}^2, i \neq j, V_i \cap V_j = \phi$;
- the union of all the elements of P_k is equal to V :
  $\cup_{i=1}^{k} V_i = V$

The elements V_i of P_k are called the *parts* of the partition. The number k is called the *cardinality* of the partition, or the number of parts of the partition.
A *balance constraint* demands that all blocks (partitions) have about equal weights. More precisely, it requires that, $\forall i \in \{1, ..., k\} :\mid V_i \mid \leq L_{max} := (1 + \epsilon)\lceil \mid V \mid / k \rceil$ for some imbalance parameter $\epsilon \in \mathbb{R}_{\geq 0}$. A block V_i is overloaded if $\mid V_i \mid > L_{max}$. The general graph partitioning problem consists in finding the partition P_k that minimizes f .

2.3 Hardness

Typically, graph partition problems fall under the category of NP-hard problems. Solutions to these problems are generally derived using heuristics and approximation algorithms.
In October of 1973 Hyafil and Rivest showed in [44] that the decision version of Graph Partitioning is hard:

Theorem 2.3.1 *(Hyafil and Rivest 1973 [44]). The problem of answering if an undirected, edge-weighted graph G can be partitioned into blocks of at most r vertices with a total weight of cut edges less than or equal to W is NP-complete.*

Andreev and Racke [1] have shown that there is no constant-factor approximation for the perfectly balanced version ($\epsilon = 0$) of this problem on general graphs. If $\epsilon \in (0, 1]$, then an $O(log^2 n)$ factor approximation can be achieved.

Theorem 2.3.2 *(Andreev and Racke [1]). For $k \geq 3$ the perfectly balanced partitioning problem has no polynomial time approximation algorithm with finite approximation factor unless P = NP.*

Also the problem is NP-hard if the block weights are constrained by $|V_i| \geq \alpha n^a$ for some $\alpha, a > 0$ or if $|V_i| = \frac{n}{2}$. The case $|V_i| \geq \alpha \log n$ for some $\alpha > 0$ is open.
In case $|V_i| \geq \alpha n^a$ also implies that the general graph partition problem with similar lower bounds on the block weights is NP-hard.

2.4 Methods of partition graph

In this section, we discuss the algorithms and methods that work with the entire graph and compute a solution . These algorithms are often used for smaller graphs or are applied as subroutines in more complex methods such as local search or multilevel algorithms.

2.4.1 Exact methods

There is a large amount of literature on methods that solve the graph partitioning problem optimally. This includes methods dedicated to the bi-partitioning case and some methods that solve the general graph partitioning problem. Most of the methods rely on the branch-and-bound framework. Bounds are derived using various approaches: Karisch et al. [47] and Armbruster et al. [2] use semi-definite programming, and Sellman et al. [70] and Sensen [71] employ multi-commodity flows. Linear programming is used by Brunetta et al. [11], Ferreira et al. [33] and by Armbruster et al. [3], and quadratic programming is adopted by Hager et al. [41]. Felner et al. [32] and Delling et al. [23, 24] utilized combinatorial bounds. Depending on the method used, the bounds derived can be very good and yield small branch-and-bound trees, but are hard to compute or the bounds are somewhat weaker and yield larger trees but are faster to compute. The latter is the case when using combinatorial bounds. On finite connected subgraphs of the two-dimensional grid without holes, the bi-partitioning problem can be solved optimally in $O(n^4)$ time [31].

All of these methods can typically solve only very small problems while having very large running times, or if they can solve large bi-partitioning instances using a moderate amount of time [23, 24], they highly depend on the bisection width of the graph. Methods that solve the general graph partitioning problem [33, 71] have immense running times for graphs with up to a few hundred nodes. Moreover, the experimental evaluation of these methods only considers small block numbers $k \leq 4$.

2.4.2 Heuristic methods

It is an approach towards finding solutions in a reasonable amount of time. Traditional methods are not performing well over large space to give results in a reasonable amount of time. Heuristic approach does not give guarantees as to always find optimal solution. Many real world problems like NP-hard can be solved by using heuristic approach. The results that are found by good approximate solutions are often NP-hard. In spite of these negative results, multiple methods for finding good approximate solutions for several restricted classes of graphs have been developed

over the years. Lipton and Tarjan [55] proved the separator theorem for planar graphs, which resulted in polynomial-time approximation schemes for several combinatorial problems, which remain NP-hard even restricted to planar graphs.
In general, there are two broad categories of methods in this domain: local and global.

Local Methods

In general, given a partition of a graph, a *local search algorithm* aims to improve an objective function (such as the number of edges that run between blocks) by moving nodes between the blocks. A local search algorithm explores solutions in the neighborhood of the current solution that is within the solution space, trying to find a better solution. There are two commonly used methods of algorithms in this approach:

(i) **Kernighan-Lin algorithm :**
Kernighan and Lin [49] were probably the first that defined the graph partitioning problem and worked on local improvement methods for this problem. The main idea of Kernighan and Lin was that, { given a balanced partition of a graph into two blocks V_1 and V_2, there are subsets $A \subset V_1$ and $B \subset V_2$ such that the partition that is created when moving the nodes in A to V_2 and the nodes in B to V_1, is globally optimal }. Kernighan and Lin then contributed a method to find *good* sets A and B to reduce the cost of a partition.

(ii) **Fiduccia-Mattheyses algorithm :**
Over time there have been many improvements made to the Kernighan-Lin algorithm. The most important improvement is a slight modification of the algorithm and the reduction in running time that was provided by Fiduccia and Mattheyses [34] in 1982. They reduced the complexity for a single pass to O(m) by using novel data structures. Like the Kernighan-Lin method, the Fiduccia-Mattheyses method performs passes in which each node is moved at most once, and the best bisection observed during an iteration (if the corresponding reduction in the number of edges cut is positive) is used as input for the next iteration.

However, instead of selecting pairs of nodes, the Fiduccia-Mattheyses method selects single nodes for movement.

Global methods

In this part, we discuss of the algorithms with methods that work with the entire graph and compute a solution directly. These algorithms are often used for smaller graphs or are applied as subroutines in more complex methods such as local search or multilevel algorithms. Many of these methods are restricted to bi-partitioning but can be generalized to k-partitioning, for example by recursion.

(i) **Spectral Partitioning :**
One of the first methods to split a graph into two blocks, spectral bisection, is still in use today. Spectral bisection infers global information of the connectivity of a graph by computing the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix L of the graph. This eigenvector z is also known as *Fiedler vector*; which is the solution of a relaxed integer program for cut optimization. A partition is derived by determining the median value m in z and assigning all nodes with an entry smaller or equal to m to V_1 and all others to V_2.

(ii) **Graph Growing :**
The second method for obtaining a bisection of a graph is called graph growing. Its simplest version works as follows. Starting from a random node v, the blocks are assigned using a breadth-first search starting at v. All nodes touched during the breadth first search are assigned to block V_1. The search is stopped after half of the original node weights are assigned to this block and V_2 is set to $V \setminus V_1$.
There are two variations of this algorithm. An algorithm called **greedy** graph growing that takes the resulting cut into account. The other variation is called **pseudo peripheral nodes**.
Since the *greedy* algorithm will be use in the next chapters , we shall explain it in more detail.

Greedy Algorithm :

A greedy algorithm, as the name suggests, **always makes the choice that seems to be the best at that moment**. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally optimal solution.
The greedy method is quite powerful and works well for a wide range of problems, it does not always yield optimal solutions, but for many problems it does.
Many algorithms can be viewed as applications of the greedy method, including minimum-spanning-tree algorithms , Dijkstra′s algorithm for shortest paths from a single source and Chvatal′s greedy set-covering heuristic. Minimum spanning-tree algorithms are a classic example of the greedy method.
The *Standard Greedy* algorithm consists of placing two random vertices into the two sets of the partition. Subsequently, the vertices are added alternately to the two sets. At each stage the vertex added is the one which results in minimum increase in cut size with ties between vertices being resolved.

(iii) **Flows :**
The algorithms are often used as a subroutine to solve related max-flow (min-cut)problems. The main idea depand on using two separate node sets in a graph by computing a maximum flow and hence a minimum cut between them. We shall use this approach in our algorithms in the next chapters.

(iv) **Geometric Partitioning :**
In this approach, we draw the graph in some geometric space (such as the unit disk in two-dimensional Euclidean space $\mathbb{R}^2$), such that the average length of an edge is short (i.e. the distance between its endpoints is small) while the points are spread out. More precisely, we require that the distance between the average pair of vertices is a fixed constant, say 1, while the distance between the average adjacent pair of vertices is as small as possible. To do this method, we need first an embedding of the graph in the geometric space.

(v) **Streaming Graph Partitioning (SGP):**
Streaming partitioning is the process of partitioning a graph in a single sweep,

reading vertices and edges only once. Thus, we incur $O(|V| + |E|)$ memory access, storage and run time, with minimal overhead.

Offline graph partitions require the entire graph to be represented in memory, whereas streaming graph partitioning may process vertices as they arrive. Streaming data models are among the most popular recent trends in big data processing. In these models the input arrives in a data stream and has to be processed on the fly using much less space than the overall input size. SGP algorithms are very fast. They are even faster than multilevel algorithms but give lower solution quality.

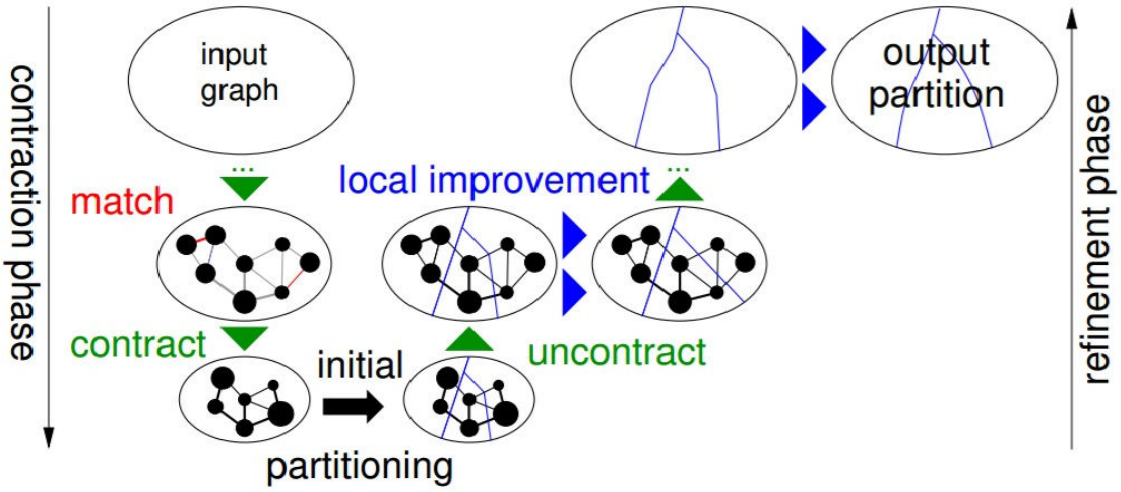


Figure 2.1 – The multilevel approach to graph partitioning.

2.5 Multilevel Graph Partitioning

The most successful heuristic for partitioning large graphs is the multilevel graph partitioning approach. The basic idea of multilevel graph partitioning can be traced back to multigrid solvers for solving systems of linear equations. More recent practical methods are mostly based on mostly graph theoretic aspects, in particular, edge contraction and local search. The graph G can be bisected using a multilevel algorithm.

The basic structure of a multilevel algorithm is very simple. The graph G is first coarsened down to a few hundred vertices, a bisection of this much smaller graph is computed, then this partition is projected back towards the original graph (finer graph), by periodically refining the partition. Since the finer graph has more degrees of freedom, such refinements usually decrease the edge-cut.

Formally, a multilevel graph bisection algorithm works as follows: consider a weighted graph $G = (V, E)$, with weights both on vertices and edges. A multilevel graph bisection algorithm consists of the following three phases. The three main phases outlined in Figure (2.1) {coarsening, partitioning, and uncoarsening and refinement}.

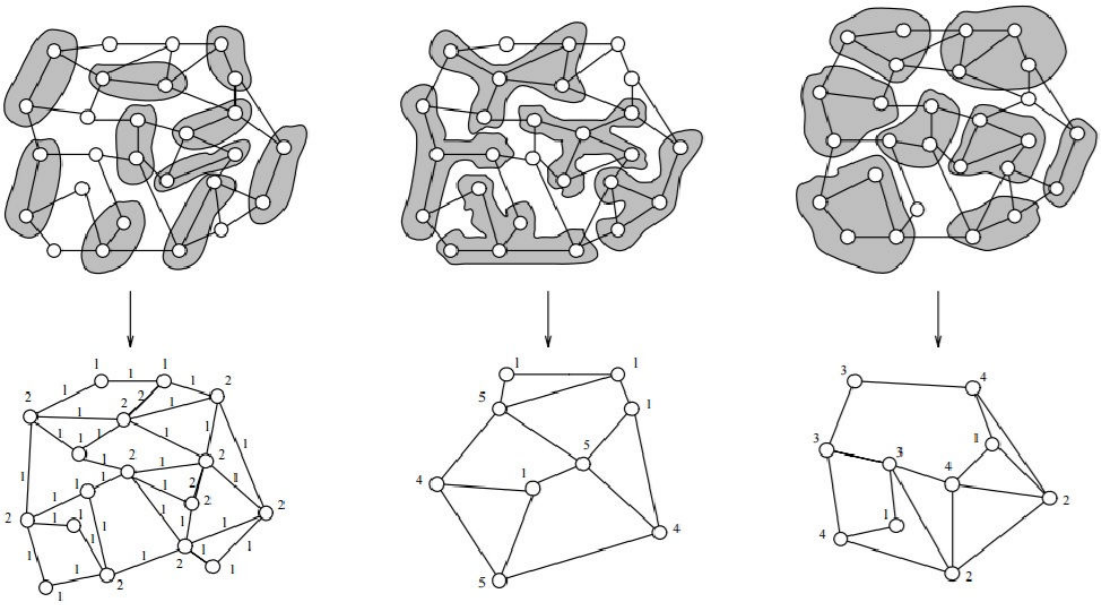


Figure 2.2 – Different ways to coarsen a graph.

2.5.1 Coarsening Phase

The main goal of the coarsening (in many multilevel approaches implemented) phase is to gradually approximate the original problem and the input graph with fewer degrees of freedom. The graph G is transformed into a sequence of smaller graphs $G_1, G_2, ..., G_m$, such that $|V_0| > |V_1| > ... |V_m|$. This phase need to a balance to determine the maximum size of sub graphs.The coarsening phase of the multilevel method can give a global outline of the graph. In this phase graph coarsening can be achieved in various ways. Some possibilities are shown in Figure (2.2).

2.5.2 Partitioning Phase

This phase creates a partition P_k^m of the graph G_m in k parts. To do so, the graph G_m, resulting from the coarsening phase, is partitioned by using a partitioning heuristic such as spectral bisection, geometric bisection, combinatorial methods. Since the size of the coarser graph G_m is small, thus this step takes a small amount of time.

2.5.3 Uncoarsening and Refinement Phase

During the uncoarsening phase, the partition of the coarsest graph G_k is projected back to the original graph by going through the graphs $G_{k-1}, G_{k-2}, ..., G_1$. Furthermore, even if the partition of G_i is at a local minimum, the partition of G_{i-1} obtained by this projection may not be at a local minimum. Hence, it may still be possible to improve the partition of G_{i-1} obtained by the projection by using local refinement heuristics. For this reason, a partition refinement algorithm is used.
Numerous refinement methods exist; however, all the refinement algorithms have two characteristics in common: (a) they have to start from an existing partition and (b) the optimization has to be local. Many of these methods are based on the Kernighan-Lin algorithm or the Fiduccia-Mattheyses variant in particular and can easily be modified to improve the balance at the same time by adding a term describing the imbalance into the cost-function used to find the next vertex to move. However, other methods exist, such as, simulated annealing, in addition, other metaheuristics have also been used as refinement methods .

2.6 Families of Graph Partitioning Problems

In this section, we recall types of families of graph partitioning problems depend on remove of vertices or edges from a graph so as to disjoint components with specific properties.

2.6.1 The k-way vertex cut problem

Given a graph $G = (V, E)$ and a positive integer $k \geqslant 2$, the *k-way vertex cut problem* is to find a partition (subset) $S \subset V$ that is minimum cardinality (min $\mid S \mid$) such that, a subgraph $G^*(V \setminus S, E)$ has at least k disjoint connected components. In this problem, a feasible solution S exists iff G has an independent set of size k. Therefore, if k is part of input,even finding a feasible solution is NP-hard. In general, this problem is NP-hard. Marx and Razgon [61] introduced a way by fixed-

parameter tractable parameterized by the size p can be solved in time $2^{O(p^3)}.n^{O(1)}$ and they showed that the problem is to be $W[1]$-hard parameterized by the size of the cut set. Berger et al [7] studied the complexity of this problem and introduced an efficient polynomial-time approximation algorithm for the problem on planar graphs. They also showed that k-way vertex cut is polynomially solvable on graphs of bounded treewidth and fixed-parameter tractable on planar graphs with the size of the separator as the parameter.

2.6.2 The k-separator problem

Given a vertex-weighted undirected graph $G = (V, E, w)$ and a positive integer k, the *k-separator problem* consists of finding a minimum-weight subset of vertices whose removal leads to a graph where the size of each connected component is less than or equal to k. If $k = 1$, the problem becomes the classical vertex cover problem. If $k = 2$ and unit weights is equivalent to computing the dissociation number of a graph.
This problem, in general, is NP-hard [74]. Spiksma et al. [72] proposed an extended formulation for the problem with some polyhedral results. Ben-Ameur et el. [9] showed that the problem can be solved in polynomial time for some graph classes (cycles and trees) by a dynamic programming approach and by using a peculiar graph transformation coupled with recent results from the literature for mK_2-free, (G_1, G_2, G_3, P_6)-free, interval-filament, a steroidal triple-free, weakly chordal, interval and circular-arc graphs. They also presented some approximation algorithms. Lepin [54] introduced an algorithm to solve this problem on polynomial time on a graph G that is modular decomposable into $\pi(G) \subseteq \{P_4, ..., P_m\} \cup \{C_5, ..., C_m\}$ in an $O(n^2)$ time, and solving on series-parallel graphs in an $O(n)$ time.

2.6.3 The vertex separator problem

Given a connected graph $G = (V, E)$ and an integer $\beta(n) < n = |V|$, find a partition of V into disjoint subsets A, B, C such that there is no edge between A and B, $|A| \leq \beta(n)$, $|B| \leq \beta(n)$, and $|C|$ is minimized. We will explain it in more detail in

the next chapter.

2.6.4 The multi-terminal vertex separator problem

Given a graph $G = (V \cup T, E)$ with $V \cup T$ the set of vertices, where T is a set of vertices called *terminals*, and E is a set of edges, the *multi-terminal vertex separator problem* (MTVSP) or also called *vertex multi-terminal cut problem* consists in partitioning $V \cup T$ into $k+1$ subsets $\{S, V_1, ..., V_k\}$, such that the size of S is minimum, each subset V_i contains exactly one terminal and no vertex in V_i is adjacent to a vertex in V_j. The MTVSP is a variant of the *k-separator problem*.
This problem can be solved in polynomial time when $\mid T \mid = 2$, and it is NP-hard when $\mid T \mid \geq 3$ [8]. Many authers researched on this topic and many algorithem were introduced. In [57], they characterized the convex hull of the solutions of this problem in two classes of graph which they called "star trees" and "clique stars". They also gave TDI systems for the problem in these graphs. In [58], they proposed three extended formulations for the problem, and they developed Branch-and-Price algorithms for the two first formulations and a Branch-and-Cut-and-Price algorithm for the third one. For more details see [56].

2.6.5 The Multiway Cut problem

This problem is also known as the *Multiterminal Cut Problem*. Given an undirected graph, $G = (V, E)$, edge weights, $w : E \longrightarrow \mathbb{R}^+$, and a set of terminals $S = \{s_1, s_2, ..., s_k\} \subseteq V$, a *multiway cut* is a set of edges that leaves each of the terminals in a separate component. The goal of the *Multiway Cut Problem* is to find a minimum weight set of edges $E^* \subseteq E$ such that removing E^* from G separates all terminals. In other words, no connected component of $G(V, E - E^*)$ contains two terminals from S. The Multiway Cut Problem is precisely finding the minimum $s-t$ cut when there are only two terminals and thus can be computed efficiently. In the case where there are three or more terminals in S, Multiway Cut is NP-Hard. When $k \geq 3$ terminals, Multiway Cut is APX-Hard, meaning that there is a constant $\delta > 1$ such that it is NP-Hard to even approximate the solution to within a

ratio of less than δ. Multiway Cut can be solved exactly for fixed k in planar graphs .

2.6.6 Vertex multicut problem

Given a graph $G = (V, E)$, a collection H of pairs of vertices $H \subset V \times V$, called terminals, and an integer $k \geq 0$. The Unrestricted Vertex Multicut (UVMC) is finding a subset V^* of V with $\mid V^* \mid \leq k$ whose removal separates each pair of vertices in H. The vertices appearing in the vertex pairs in H are called *terminals*. Where, the Restricted Vertex Multicut (RVMC) is finding a subset V^* of V with $\mid V^* \mid \leq k$ that contains no terminal and whose removal separates each pair of vertices in H. Calinescu et al. [14] showed that RVMC is NP-complete in bounded-degree trees and the easier UVMC is polynomially solvable in trees but becomes NP-complete in bounded-degree graphs of treewidth two. Moreover, they gave a Polynomial-Time Approximation Scheme (PTAS) for UVMC in graphs of bounded treewidth. Papadopoulos [66] proposed a polynomial-time algorithm for the problem on permutation graphs. They showed that the problem remains NP-complete on split graphs whereas it becomes polynomial-time solvable for the class of co-bipartite graphs .

2.6.7 Critical Nodes Problem (CNDP)

Given a graph $G = (V, E)$ and an positive integer k, the critical nodes problem e is to find a subset $A \subseteq V$ of nodes such that $\mid A \mid \leq k$, whose deletion minimizes the pair-wise connectivity among the nodes in the induced subgraph $G(V \setminus A)$. Arulselvan et al.[4] showed that the problem is NP-complete, and and derived a mathematical formulation based on integer linear programming. Ventresca and Aleman [73] proposed an algorithm based on a modified depth-first search that requires $O(k(\mid V \mid + \mid E \mid))$ time complexity. Also, they employed the method within in a greedy algorithm for quickly identifying R. Di Summa et al.[29] proposed an integer linear programming model with a non-polynomial number of constraints but whose linear relaxation can be solved in polynomial time. Also, they proposed an alternative model based on a quadratic reformulation of the problem.
The CNDP has applications in many fields including social network analysis, qua-

lity assurance and risk management in telecommunication networks, transportation science, and control of social contagion. Therefore, many authors and researches have discussed this problem.

2.6.8 The k-way edge cut problem

This problem is also called The *k-Cut Problem.* We can describe it by giving an undirected graph, $G = (V, E)$ and edge weights $w : E \longrightarrow \mathbb{R}^+$, and an integer k. The goal of the k-Cut Problem is to find a minimum weight set of edges $E^* \subseteq E$ such that removing E^* from G leaves k connected components. The k-Cut Problem is precisely finding the global minimum $s-t$ cut when $k = 2$ and is polynomial time solvable.

2.6.9 The Balanced Graph Partitioning problem

Given a graph $G = (V, E)$,the problem of $(k, \upsilon) - balanced$ graph partitioning dividing the vertices of a graph into k almost equal size components (each size is less than $\upsilon.\frac{n}{k}$), so that the capacity of edges between different components is minimized. If $k = 2$ and $\upsilon = 1$, the problem would be reduced to a minimum bisection problem. For more details on this problem, see [1].

Chapter 3

Vertex Separator problem

Contents

3.1 Introduction **42**

3.2 Formulation **44**

3.3 Neighborhood searching **45**

3.3.1 Algorithm 1 45

3.3.2 Algorithm 2 46

3.4 Computational experiments **46**

3.5 Conclusions and Remarks **53**

In this chapter, we give some basic notions on the vertex separator problem. We then propose two algorithms to solve the vertex separator problem (VSP). These algorithms are done by searching on the neighborhood of any vertex in the graph and choose the vertex has minimum neighborhood with specific property. We also present the computaional results for three sets of graphs that we take them from the matrix Market library, VSP benchmark instances from the DIMACS challenge on graph coloring, and the set of graphs generated by [43].

3.1 Introduction

The vertex separator for a graph is a subset of vertices whose removal disconnect the graph into at least two non-empty connected components. Given $G = (V, E)$ a connected undirected graph and a positive integer $\beta(n)$, where $n = |V|$, the vertex separator problem (VSP for short) is to find a partition of V into three nonempty classes A, B, and C such that:

(i) There is no edge between A and B;

(ii) $\max\{|A|, |B|\} \leq \beta(n)$;

(iii) $|C|$ is minimum.

The subsets A and B are called *shores* of the separator C. For convenience, a partition $\{A, B, C\}$ of V which satisfies (i) and (ii) will be also called a separator. Figure (3.1) show example of VSP.

The VSP is NP-hard [13, 37]. Even for special classes of structured graphs VSP remains NP-hard,(if G is planer graph[36], as well as for [39], bipartite graphs [38], grid graphs and unit disk graphs [27]).

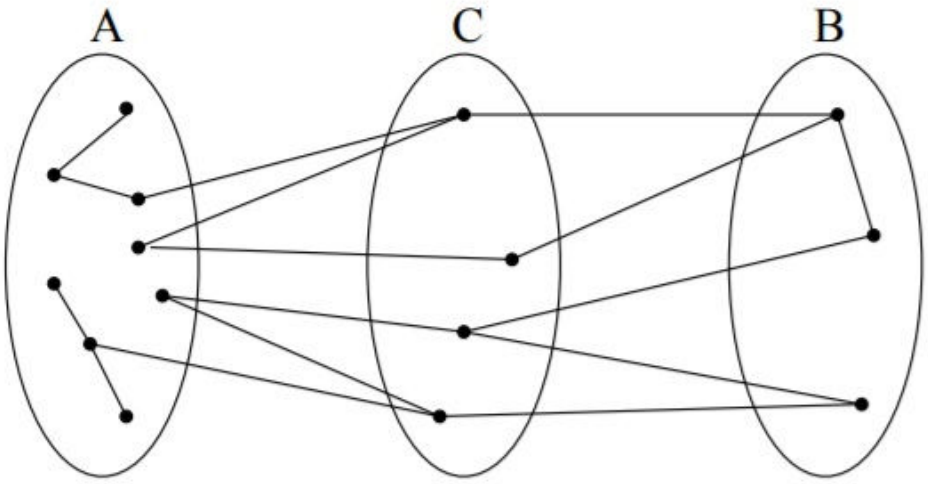


Figure 3.1 – example of VSP

When $\beta(n) = n - k$ for some positive constant k, the problem becomes polynomially solvable [5]. As it was mentionned in [5], the VSP is trivial if $\beta(n) = 1$ and it is polynomially solvable if $\beta(n) \geq n - 1$.

The VSP is useful for many graph algorithms and has a number of applications: (VLSI) design for partitioning circuits into smaller subsystems, with a small number of components on the boundary between the subsystems. The decisional version of the VSP consists of finding a vertex separation value larger than a given threshold. It has applications on computer language compiler design and exponential algorithms. In compiler design, the code to be compiled can be represented as a directed acyclic graph (DAG) where the vertices represent the input values to the code as well as the values computed by the operations within the code. (Communication networks, bioinformatics, Drawing and Natural Language Processing (see [5] for a survey of VSP applications.)).

Several exact algorithms have been proposed for solving VSP [5, 26, 28, 25, 16]. These VSP algorithms are able to find optimal results in a reasonable computing time (within 3 hours) for instances with up to 150 vertices, and may fail to solve larger instances, a variety of heuristic algorithms have been found to be very efficient in finding near-optimal partitions. Benlic and Hao [10] and Zhang and Shao[75] used breakout local search BLS for solve VSP. Jesús, Juan and Abraham[45] used four shake algorithms and embed them into a Variable Neighborhood Search scheme to solve this problem. Sanchez-Oro et al.[68] introduced several Variable Neighborhood Search(VNS) algorithms, which alternate between a local search phase and a shaking phase. Hager et al.[40] proposed a continuous optimization approach.

Heuristic and meta-heuristic methods, which have shown to be very useful for various NP-hard combinatorial optimization problems, have not been considered up until now for VSP. Although such methods have no formal performance guarantee, they have shown to be able to provide solutions of acceptable quality with reasonable computing efforts even for very large instances.

The remainder of this section is devoted to more definitions and notations.

The graphs we consider are finite, undirected and connected. We denote a graph by $G = (V, E)$, where V is the node set and E is the edge set. If e is an edge with end-nodes u and v, then we write $e = (uv)$. If $W \subseteq V$, the set of edges having

one end-node in W and the other one in $\overline{W} = V \setminus W$ is called a *cut* and is denoted by $\delta(W)$. The set of edges having both end-nodes in W will be denoted $E(W)$. If W_1, W_2 are disjoint subsets of V, then $[W_1, W_2]$ denotes the set of edges of G which have one node in W_1 and the other one in W_2. For $U \subseteq V$ we denote by $G(U)$ the induced subgraph on U (i.e., $G(U) = (U, E(U))$. If $F \subseteq E$, then $V(F)$ denotes the set of nodes of F and $G(F)$ the subgraph of G induced by F.

3.2 Formulation

Let $G = (V, E)$ be an undirected graph and $\beta(n)$ be a positive integer, where $n = |V|$. For every separator $\{A, B, C\}$ we associate an incidence vector $(x, y) \in \mathbb{R}^{2n}$ defined by $x_v = 1$ if $v \in A$ and 0 otherwise, and $y_v = 1$ if $v \in B$ and 0 otherwise.

The VSP is equivalent to the integer linear program:

$$max\{\sum_{v \in V}(x_v + y_v)$$

s.t.

$$x_u + y_v \leq 1 \qquad \forall (uv) \in E \qquad (3.2.1)$$

$$x_v + y_u \leq 1 \qquad \forall (uv) \in E \qquad (3.2.2)$$

$$x_v + y_v \leq 1 \qquad \forall v \in V \qquad (3.2.3)$$

$$1 \leq \sum_{v \in V} y_v \leq \beta(n) \qquad (3.2.4)$$

$$1 \leq \sum_{v \in V} x_v \leq \beta(n) \qquad (3.2.5)$$

$$x_v \in \{0, 1\} \qquad \forall v \in V \qquad (3.2.6)$$

$$y_v \geq 0 \qquad \forall v \in V \qquad (3.2.7)$$

First and second inequalities come from the fact that there is no edge between A and B. Third inequalities come from the fact that $A \cap B = \emptyset$. Fourth and fifth inequalities come from the fact that $A \neq \emptyset \neq B$ and $\max\{|A|, |B|\} \leq \beta(n)$.

Let u and v be tow nonadjacent nodes. Denote by α_{uv} the maximum number of node-disjoint paths between u and v. Define

$$\alpha_{min} = min\{\alpha_{uv} : u, v \in V, uv \in E\}.$$

We note that for any path between any two vertices $a \in A$ and $b \in B$, then C have at least one vertex in common. Therefore $|C| \geq \alpha_{min}$. Thus, we use α_{min} is as a lower bound of the cardinality of any separator.

3.3 Neighborhood searching

In a graph $G = (V, E)$, two vertices u,v are considered as neighbors if uv is an edge in E. The neighborhood of a vertex u, denoted by $N(u)$, is the set of vertices $v \in V$ such that $uv \in E$. The neighborhood of a subset $A \subseteq V$, denoted by $N(A)$, is the set of vertices $v \in V \setminus A$ such that $uv \in E$ for some $u \in A$.

3.3.1 Algorithm 1

In this algorithm, we order the vertices of V by increasing degree, and then choose the first vertex in the order and put it in the set A and find every other vertex connected with it (neighborhood of it), put these vertices in set $C = N(A)$ and the rest in set B, and then check the condition $|A| + |C| \geq n - \beta$ if satisfy, stop, else continue, where n is number of vertices and $\beta = \lfloor \frac{2 * n}{3} \rfloor$ gives in an instance .
The goal of this algorithm is to find a good lower bound for VSP in short time.

Details our algorithm for VSP.

Algorithm 1 algorithm 1

1: *order the vertices by increasing degree.*
2: $A = \phi; C = \phi; B = \phi$.
3: *Let* $v = V \setminus (A \cup C)$*with a minimum degree,* $A = A \cup \{v\}; C = N(A); B = V \setminus (C \cup A)$.
4: **if** $|A| + |C| \leq n - \beta$ **then goto** *3*
5: **else** *stop.*

3.3.2 Algorithm 2

In this algorithm, we choos the vertex has minimum degree in the first step and put it in the set A and find the neighborhood of it and put them in set C and the rest of vertices in the set B. In the second step choos a vertex i not in A with minimum neighborhood in $B \setminus \{i\}$. Put i in the set A, and so on until satisfy the condition $|A| + |C| > n - \beta$.

We can use the result of this algorithm as a lower bound when we solve the problem integrality. Also we get a new α_{min} called $(\alpha*_{min})$ greater than or equal the originalα_{min}.

we provide details our algorithm for VSP.

Algorithm 2 algorithm 2

1: *choose the vertex a that has minimum degree.(if there are more than one, choose any one).* $A = \{a\}; C = \phi; B = \phi$.
2: $C = N(A); B = V \diagdown (C \cup A)$.
3: **while** $|A| + |C| \leq n - \beta$ **do**
4: *Let* $i \in V \setminus A$ *such that* $|N(i) \cap B|$ *is minimum* .
5: $A = A \cup \{i\}; C = N(A); B = V \setminus (A \cup C)$.
6: *End while*

Now, let us call $(|A^*|, |B^*|, |C^*|)$ be a solution of our algorithm, we will try finding a solution better than $(|A^*|, |B^*|, |C^*|)$, in other word, a solution $|A + B|$ such that $|A+B| > |A^* + B^*|$, to do this we will add some inequalities as cuts to our problem. The details in the next section.

3.4 Computational experiments

Our algorithms 1 and 2 are programmed in C++ and compiled with GNU g++ under GNU/Windows 7 running on a Dell laptop with an Intel Core i7-4800MQ with 2.70 GHz and 8 GB of RAM. The 11 instances in the first class, called DIMACS graphs, come from the DIMACS challenge on graph coloring. The graphs included have 450 vertices.These instances can be download from HTTP://`www.ic.unicamp.br/~cid/Problem-instances/VSP.html`.

In the second class, a set of 54 more challenging graphs generated by Helmberg and Rendl [43]. This benchmark consists of toroidal, planar, and random graph instances ranging from 800 to 3000 vertices. For our experiments on these graphs, all the vertices are given unit weights. These instances can be download from HTTP://`www.optsicom.es/maxcut/instances`.

They have been conducted in two steps. We first calculated α_{min}, the lower bound of any cardinality of any separator. We used the code developed by Cherkassky and Goldberg [18] for the maximum flow problem(Source code available at [77]). We then solved the following mixed-integer program using Ilog-CPLEX 12.6 [78].

$$\text{Max} \sum_{v \in V} (x_v + y_v)$$

subject to

$$x_u + y_v \leq 1 \quad , \quad x_v + y_u \leq 1 \qquad \forall (uv) \in E \tag{3.4.1}$$

$$x_v + y_v \leq 1 \qquad \forall v \in V \tag{3.4.2}$$

$$\sum_{v \in V} (x_v + y_v) \leq n - \alpha_{min} \tag{3.4.3}$$

$$1 \leq \sum_{v \in V} x_v \leq \lfloor \frac{n - \alpha_{min}}{2} \rfloor \tag{3.4.4}$$

$$1 \leq \sum_{v \in V} y_v \leq \beta(n) \tag{3.4.5}$$

$$1 \leq \sum_{v \in V} x_{ia} \leq \beta(n) \tag{3.4.6}$$

$$x_v, y_v \in \{0, 1\} \qquad \forall v \in V \tag{3.4.7}$$

The inequality $\sum_{v \in V} x_v \leq \lfloor \frac{n - \alpha_{min}}{2} \rfloor$ comes from the following:

let $\{A, B, C\}$ be a separator. We can suppose without loss of generality $|A| \leq |B|$. Since $|A| + |B| \leq n - \alpha_{min}$, we have $|A| \leq \lfloor \frac{n - \alpha_{min}}{2} \rfloor$.
Scince the graphs have height orders, then they take more time and more branch nodes to solve our problem in Mixed-integer programming; then, we need to impro-

vement some inequalities. Therefore, we replace the inequality

$$\sum_{v \in V}(x_v + y_v) \leq n - \alpha_{min}$$

by

$$\sum_{v \in V}(x_v + y_v) \leq n - \alpha*_{min} \tag{3.4.8}$$

where $\alpha*_{min} \geqslant \alpha_{min}$.
This $\alpha*_{min}$ is as result from our algorithms by arranging every $\alpha_i, i \in V$ in increasing order and choosing α_i has order $|A| + 1$, where $|A|$ is a result from our algorithm.

The results of our computational experiments are presented in Tables 1,2,3 and 4, where each line corresponds to one specific instance.

In table 3.1, we select a subset of instances taken from [76] as the same instances taken from [16] so as to compare with it and to know behavior the algorithms with approach to the optimal solution, where table 1 consist seven columns. The first one represents the instance name. The second is the number of vertices of graph. The third is the result of algorithm 2. The fourth is the optimal solution by solving a mixed integer program. The fifth is the time for the optimal solution, the sixth is the results for applaying Lagrangian relaxiation and cutting planes by [16]. The final column is the time for algorithm by [16] . The time to find the solution by our algorithm for all of the instances is less than 0.05 seconds.

The algorithm by [16] failed to solve graphs higher of 250 vertices.

In table 3.2, there are 8 columns. The first column contains the instance name where all instances have 450 vertices and $\beta = \lfloor \frac{2 * n}{3} \rfloor$. The second column contains the value of α_{min}. The third column contains the value of $\alpha*_{min}$. The fourth column contains the results of algorithm 1. The fifth column contains the time of finding the results by algorithm 1 in seconds. The sixth column contains the results of algorithm 2. The seventh column contains the time of finding the results by algorithm 2 in seconds. The final column contains the upper bound when we use the best results between algorithm 1 and 2 as cuts when we use mixed-integer program (time limit

Table 3.1 – Compare our results with 25 instances

Instances	$\|V\|$	***algorithm 2***	*opt.*	*time*	*Lagrang.*	*time Lag.*
miles1000	128	110*	110	1.87	109	3.09
DSJC125.9	125	22*	22	0.58	22	4.4
L125.fs-183-1	125	92	98	1.89	96	2.04
bcsstk04	132	83	84	4.85	84	3.74
arc130	130	88*	88	4.54	88	5.5
L120.cavity01	120	95	99	1.22	99	3.24
L120.fidap021	120	93	98	2.75	98	2.13
L120.rbs480a	120	87	88	4.03	88	2.34
L100.rbs480a	100	72	73	0.90	73	2.3
L120.e05r0000	120	85	90	2.67	90	2.82
L100.wm1	100	74*	74	1.20	73	1.63
L120.fidap022	120	84*	84	2.56	84	2.95
L100.wm2	100	76*	76	1.25	76	1.69
L100.fidapm02	100	69*	69	2.01	69	2.04
L120.fidap001	120	82*	82	3.54	82	3.76
L100.e05r0000	100	69	70	1.12	70	1.81
L80.fidapm02	80	53*	53	0.48	53	1.21
L120.fidapm02	120	85	86	1.92	86	2.96
L100.fidap001	100	64*	64	1.12	64	2.62
L100.fidap022	100	62*	62	1.36	62	2.37
L80.fidap022	80	41*	41	0.34	41	1.81
L100.fidap027	100	69*	69	1.69	69	2.55
L100.fidap002	100	66*	66	1.19	66	2.52
L120.fidap002	120	67	68	1.72	68	3.58
L120.fidap027	120	83*	83	1.84	83	3.74

* the result from our algorithm is equal to an optimal solution.

by 3 hours). The cuts as follows:

$$|A| \geqslant t \tag{3.4.9}$$

$$|A| + |B| \geqslant \beta + t \tag{3.4.10}$$

$$|A| + |B| \leq n - \alpha*_{min} \tag{3.4.11}$$

where n is number of vertices of graph and t is the greater number of vertices in A that we have it from algorithm 1 or 2. We note that for any instance if $\beta + \alpha_{min} > n - 1$, then we can not add a condition(3.4.10) to the problem.

In table 3.3, there are 8 columns. The first column contains the name of the instance. The second column contains the number of vertices of the graph *(number of vertices)* where the number of vertices of the graph is between 800 to 2000. The third column contains the results of algorithm 2, note that we use the results of

Table 3.2 – DIMACS instances (11 instances)

Instances	α_{min}	$\alpha*_{min}$	*t1*	*time t1*	*t2*	*time t2*	*UB*
le450-5a	13	16	317	0.10	315	0.06	326
le450-5b	12	16	316	0.11	314	0.13	326
le450-5c	27	31	308	0.20	307	0.07	318
le450-5d	29	32	309	0.61	309	0.10	319
le450-15b	1	12	326	0.72	325	0.14	336
le450-15c	18	32	306	0.11	307	0.10	317
le450-15d	18	33	304	0.16	308	0.14	318
le450-25a	2	6	340*	0.62	338	0.17	340
le450-25b	2	8	337	0.14	334	0.22	347
le450-25c	7	21	311	0.20	311	0.11	322
le450-25d	11	23	311	0.41	310	0.11	321

we note that in le450-25a finding an optimal solution after 102 second and openning 2244 nodes .

algorithm 2 only since they are best from the results of algorithm 1. The fourth column contains the time of algorithm 2 in seconds. The fifth column contains the rseults of Benlic and Hao [10] to compare with it, where they used four algorithms and we choose the best one that is BLS algorithm. The sixth column contains the time of BLS algorithm in seconds where the BLS algorithm is programmed in C++ and compiled with GNU g++ under GNU/Linux running on an Intel Xeon E5440 with 2.83 GHz and 2 GB of RAM.

Since the instances are big then if we want to find exact solution we need along time , then the best way to know the range of an optimal solution is by applying the results of algorithm 2 as a LB (Lower Bound) and we use a new tchnique to know this range in less time. This technique is to add a new cut to our problem, this cut is to replace the original α_{min} by $\alpha_{new} = 0.1 * n$ if $\alpha_{min} < \alpha_{new}$; therefore, (3.4.11)becoms

$$|A| + |B| \leq n - \alpha_{new} \tag{3.4.12}$$

The seventh column contains the value of LB for our problem after adding the cut (3.4.12) and use mixed- integer program upto 1800 seconds for the value of original β. The final column contains the value of UB for our problem with out adding the cut (3.4.12) and resolution the problem mixed- integer programming (time limit by 3 hours).

Table 3.3 – Comparision of algorithm 2 and BLS algorithms on graphs generated by Helmberg and Rendl (27 instances).

Instances	N	*algo.2*	*time*	*BLS*	*time BLS*	*LB*	*UB*
g1	800	543**	0.96	543	7.2		543
g2	800	543**	0.66	543	8.7		543
g3	800	543**	0.23	543	66.3		543
g5	800	543**	0.19	543	65.9		543
g6	800	543**	0.25	543	8.5		543
g7	800	543**	0.26	543	10.0		543
g8	800	543**	0.26	543	69.6		543
g9	800	543**	0.15	543	34.0		543
g10	800	543**	0.21	543	70.7		543
g11	800	784**	2.68	784	0.0		784
g12	800	768**	1.48	768	0.0		768
g13	800	755**	1.43	755	0.2		755
g14	800	628	1.22	654	768.7	644	680
g15	800	633	1.22	656	856.7	642	680
g16	800	626	1.17	656	188.6	646	680
g17	800	633	1.10	656	401.1	651	680
g18	800	628	1.18	654	623.5	645	680
g19	800	633	1.17	656	1085.8	645	680
g20	800	626	1.09	656	353.2	641	680
g21	800	634	1.11	656	552.5	644	680
g22	2000	1412	10.71	1412	657.7		1800
g23	2000	1410	10.40	1410	310.4		1800
g24	2000	1411	10.68	1411	371.5		1800
g25	2000	1411	10.56	1411	832.5		1800
g26	2000	1413	10.70	1413	313.9		1800
g33	2000	1950**	39.59	1950	0.2		1950

** an optimal solution for the instance.
In instances g11, g12, g13 and g33 we find the optimal solution after 2.68, 1.48, 1.43 and 1.01 seconds respectively.

In table 3.4, there are 6 columns, the first column contains the name of the instances. The second column contains the number of vertices of the graph, which is between 800 to 3000. The third column contains the results of algorithm 2. The fourth column contains the time of algorithm 2 in seconds. The fifth column contains the value of LB for our problem after adding the cut (3.4.12) and using mixed-integer program after the max time 1800 seconds. The final column contains the value of UB for our problem with out adding the cut (3.4.12) and using mixed-integer program (time limit by 3 hours).

Table 3.4 – Results of algorithm 2 on graphs generated by Helmberg and Rendl (27 instances)

Instances	N	*algo.2*	*time*	*LB*	*UB*
g4	800	543**	0.34		543
g27	2000	1412	10.18		1700
g28	2000	1410	10.29		1700
g29	2000	1410	10.52		1700
g30	2000	1410	10.60		1700
g31	2000	1411	10.36		1700
g32	2000	1960**	38.60		1960
g34	2000	1920**	38.09		1920
g35	2000	1580	32.12	1602	1700
g36	2000	1580	32.32	1609	1700
g37	2000	1576	31.91	1597	1700
g38	2000	1573	31.60	1606	1700
g39	2000	1580	32.74	1610	1700
g40	2000	1580	32.74	1609	1700
g41	2000	1576	31.6	1608	1700
g42	2000	1578	31.66	1618	1700
g43	1000	705	0.97		750
g44	1000	705	1.02		750
g45	1000	706	1.03		750
g46	1000	703	0.97		750
g47	1000	704	0.97		750
g48	3000	2900**	143.8		2900
g49	3000	2940**	145.7		2940
g50	3000	2950**	149.5		2950
g51	1000	788	2.92	813	850
g52	1000	783	2.88	812	850
g53	1000	789	2.93	810	850
g54	1000	786	2.77	813	850

** an optimal solution for the instance.
In instances g32, g34, g48, g49 and g50 we find an optimal solution after 0.95, 1.14, 14.03, 17.69 and 13.66 seconds respectively.

3.5 Conclusions and Remarks

In this chapter, we have introduced two heuristics and found results help us to know the best LB and UB in a short time possible whenever we want to find exact solutions for the vertex separator problem. Some of results we got were actually are optimal solutions. We compared our results with Cavalcante and de Souza[16], and Benlic and Hao[10]. The results we obtained for most the instances is the same with less time. Also, we obtained results for the 27 instances that have $\beta = \lfloor \frac{2 * n}{3} \rfloor$ but there were no more information to compare it to; therefore, we use them as future reference VSP approaches.

Chapter 4

st-Connected vertex separator problem

Contents

4.1 Introduction **57**
4.2 The *st*-connected separator problem **57**
4.3 Formulations **58**
4.3.1 Natural Formulation (NF) 58
4.3.2 Extended Formulation (EF) 59
4.3.3 Tree Extended Formulation (TEF) 60
4.4 Polyhedral study **61**
4.4.1 Articulation inequalities 65
4.4.2 Partition inequalities 68
4.5 Heuristic **68**
4.5.1 Computational experiments 68
4.6 Conclusions and Remarks **74**

In this chapter, we give some basic notions concerning the st-connected vertex (edges) separator problem(st-CVS(CES) problem), then we give three integer programming formulations for this problem. We investigate the related polyhedron and

discuss its polyhedral structure. We describe some valid inequality. Also, we introduce a heuristic to solve this problem, along with an extensive computational study is presented.

4.1 Introduction

Vertex and edge connectivity are two of the most fundamental concepts in network reliability theory. There are many articles that have studied a specific case, clique vertex separator, for example ([22, 53, 65]). Algorithms of this nature are popularly known as Fixed-Parameter Tractable algorithms (FPT) with parameter as the solution size. Marx et al. [60] shows that the *st*-CVS problem is NP-Hard, and it is in FPT. Narayanaswamy and Sadagopan [63] show that *st*-CVS is NP-Complete on graphs of chordality at least 5 and present a polynomial-time algorithm for *st*-CVS on chordality 4 graphs. They also introduced an algorithm to solve the problem and proved that *st*-CVS parameterized is $W[2]$-hard. Recently, this problem was studied by [56] as complexity and polyhedral point of view.

A simple undirected graph G is a pair (V, E) where V is the set of vertices and E is the set of edges. We note $n := \mid V \mid$ the number of vertices and $m := \mid E \mid$ the number of edges. Moreover, each edge $\{u, v\}$ is denoted uv. For all $S, T \subseteq V$, we put $\delta(S,T) := \{uv \in E : u \in S, v \in T\}$, $\delta(S) := \delta(S, V \setminus S)$ and $E(S) := \{ij \in E : i, j \in S\}$. For all $F \subseteq E$, we put $V(F) := \{u, v \in V : uv \in F\}$. Moreover, we put $N(S) := \{u \in V \setminus S : \exists v \in S, uv \in E\}$. For Each vertex $u \in V$, we set $\delta(u) := \delta(\{u\})$ the set of incident edges at the vertex u, $N(u) := \{v \in V : uv \in E\}$ the neighborhood of the vertex u and $d(u) := \mid \delta(u) \mid$ The degree of the vertex u. For the simple graphs we have $d(u) = \mid N(u) \mid$. For all $S \subseteq V$, we denote by $G[S]$ the graph induced by S. For all $S \subseteq V$, for all $u \in V$, we put $NS(U) := N(u) \cap S$ the neighborhood of the vertex u in $G[S]$ and $dS(U) := \mid NS(U) \mid$ The degree of the vertex u in $G[S]$.

4.2 The *st*-connected separator problem

Let s and t be two disjoint vertices of V, not adjacent. An *st*- connected separator in the graph G is a set $S \subseteq V \setminus \{s, t\}$ such that

(i) there are no more paths between s and t in $G[G \setminus S]$, and

(ii) the graph $G[S]$ is connected.

We shall assume that G is connected. Moreover, for any st-connected separator S in G, there exists at least one bi-partition (A, B) of $V \setminus S$ such that $s \in A$, $t \in B$, and $\delta(A, B) = \phi$.

4.3 Formulations

In this section, we will give some formulations.

4.3.1 Natural Formulation (NF)

The $(s,t) - CVS$ is equivalent to solving the linear programm

$$min\{\sum_{v \in V}(w_v x_v),\ x \in P_{st}(G)\}.$$

The st-connected separator problem can be formulated as follows :
$min \sum_{v \in V}(x_v)$
s.t.

$$x_s = x_t = 0 \tag{4.3.1}$$

$$X(V_I(P_{st})) \geq 1 \ \ \forall P_{st} \in \Gamma_{st}, \tag{4.3.2}$$

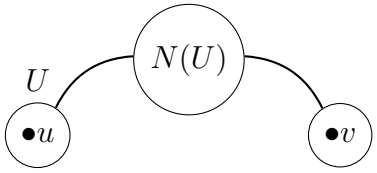
$$\begin{aligned} X(N(U)) \geq x_u + x_v - 1 \quad & \forall u, v \subseteq V \setminus \{s,t\} with\ uv \notin E, \\ & \forall U \subseteq V \setminus \{s,t\} with\ u \in U, v \notin U \cup N(U) \end{aligned} \tag{4.3.3}$$

$$x_v \in \{0,1\} \ \ \forall v \in V. \tag{4.3.4}$$

where P_{st} is a path between s and t, $V_I(P_{st}) \equiv V(P_{st} \setminus \{s,t\})$ is the set of vertices in the internal path P_{st} and Γ_{st} is the family of all paths between s and t, and $N(u)$ is the neighborhood of the vertex u.

Inequalities (**??**) mean that from any path between s and t, there exist at least one vertex from the internal path in S .
Inequalities (4.3.3) give us guarantee that the separator is connected.

Figure (4.1) shows the condition (4.3.3).



Proposition 4.3.1 *Inequalities (4.3.3) can be separated in polynomial time.*

proof : Let $x \in \mathbb{R}_+^{|V|}$. For every $w \in V$, consider a vertex cost equal to x_w. Let u,v be two vertices such that $(uv) \notin E$, we find a minimum cost (u,v)-separation $U_{u,v} \subset V$.
We suppose without loss of generality that $u \in U_{u,v}$. Then by comparing $x(U_{u,v})$ and $x_u + x_v - 1$, one can obtain the most violated inequality if there is any.

4.3.2 Extended Formulation (EF)

Let $z \in \{0,1\}^{|E|}$ be the variable vector defined by , z_e=1 if the edge e is in the graph $G[S]$, and 0 otherwise, for all $e \in E$. Then, we have the following model: EF

min $x(V)$,
s.t.

$$x_s = x_t = 0 \tag{4.3.5}$$

$$X(V_I(P_{st})) \geq 1 \ \ \forall P_{st} \in \Gamma_{st}, \tag{4.3.6}$$

$$z_e \leq x_u \ \ \forall u \in V, \ \ \forall e \in \delta(u), \tag{4.3.7}$$

$$z_e \geq x_u + x_v - 1 \ \ \forall e := uv \in E \setminus (\delta(s) \cup \delta(t)), \tag{4.3.8}$$

$$z(\delta(U)) \geq x_u + x_v - 1 \ \ \forall u, v \in V \setminus \{s,t\}, uv \notin E, \ and \ \forall U \subset V \setminus \{s,t\}, u \in U, v \notin U, \tag{4.3.9}$$

$$z_e \in \{0,1\} \ \ \forall e \in E, \tag{4.3.10}$$

$$x_v \in \{0,1\} \ \ \forall v \in V. \tag{4.3.11}$$

Note that, we can replace $z_e \in \{0,1\}$ by $0 \leq z_e$. In fact, the integrity of x implies integrity of z.
Also, according to the constraint (4.3.7), we will always have the following equality:

$$z_e = 0 \qquad \forall e \in \delta(s) \cup \delta(t) \tag{4.3.12}$$

4.3.3 Tree Extended Formulation (TEF)

Let $y \in \{0,1\}^{|E|}$ be the variable vector indicating whether an edge is in the tree in the separator to guarantee the connectivity of separator S defined by for every $e \in E$, y_e=1 if the edge e is chosen to form the covering tree of $G[S]$, and 0 otherwise. Then, we have the following model: TEF
min $x(V)$,

s.t.

$$x_s = x_t = 0 \tag{4.3.13}$$
$$x(V(P_{st})) \geq 1 \ \ \forall P_{st} \in \Gamma_{st}, \tag{4.3.14}$$
$$y_e \leq x_u \ \ \forall u \in V, \forall e \in \delta(u), \tag{4.3.15}$$
$$y(E) = \mid V \mid -1, \tag{4.3.16}$$
$$y(\delta(U)) \geq x_u + x_v - 1 \qquad \forall u, v \in V \setminus \{s,t\}, uv \notin E, \ and \ \forall U \subset V, u \in U, v \notin U, \tag{4.3.17}$$
$$0 \leq x_v \leq 1 \ \ \forall v \in V, \tag{4.3.18}$$
$$0 \leq y_e \leq 1 \ \ \forall e \in E, \tag{4.3.19}$$
$$x_v \in \{0,1\} \ \ \forall v \in V. \tag{4.3.20}$$

Proposition 4.3.2 *: Formulation TEF dominates the formulation EF.*

Proof: Let (x,y) be a solution in the polyhedron $P^y(G)$, the polyhedron associated with the formulation TEF. Let us assume $z_e := max\{y_e, x_u x_v\}$ for any edge $e := uv \in E$. We show that the solution (x,z) is a solution in the polyhedron $P^z(G)$, the polyhedron associated with the formulation EF.

- The constraints (4.3.5) and (4.3.6) are trivially verified because the solution x satisfies constraints (4.3.13) and (4.3.14).
- The constraint (4.3.7) is satisfied by this solution. Thus, we have the two following cases:
  - Let $z_e = y_e$. Now, the constraint (4.3.15) is satisfied. Hence, we have $z_e \leq x_u$.
  - Let $z_e = x_u x_v$. Now, we have $x_v \leq 1$. Hence, we have $z_e \leq x_u$.
- The constraint (4.3.8) is satisfied by this solution. Indeed, by definition of z, we have $x_u x_v \leq z_e$. Now, since $0 \leq x_u \leq 1$ and $0 \leq x_v \leq 1$, we have $(1-x_u)(x_v-1) \leq 0$. Then, we have $(1-x_u)(x_v-1) \leq 0 \Longleftrightarrow x_u+x_v-1-x_u x_v \leq 0 \Longleftrightarrow x_u + x_v - 1 \leq x_u x_v$. So, we have $x_u + x_v - 1 \leq z_e$.
- The constraint (4.3.9) is satisfied by this solution. Indeed, we have $y_e \leq z_e$ for all $e \in E$, Hence, $y(\delta(U)) \leq z(\delta(U))$. Consequently, according to the constraint (4.3.17), we have the result.
- The constraint (4.3.10) is satisfied by this solution. Indeed, we have $z_e \geq 0$ because $y_e \geq 0$ and $x_u x_v \geq 0$, and $z_e \leq 1$ because $y_e \leq 1$ and $x_u x_v \leq 1$.$\boxdot$

4.4 Polyhedral study

The convex hull of the incidence vectors of all separators is called the connected (s,t) vertex separator polyhedron and denoted $P_{st}(G)$, that is,

$$P_{st}(G) = \text{conv}\{x^S : S \,\text{is a}\, st - \text{connected separator }\}$$

.

Consider S is an st-connected separator in G. Let us denote by $C_1, ..., C_k$ the sets of vertices corresponding to the connected components of $G[V \setminus \{s,t\}]$. Let us show that one can always reduce to the case where k = 1.

- Let us prove that the following condition is a necessary condition for the existence of a separator of the vertices s and t:

There exists a unique connected component $G[C_{i^*}]$ *of* $G[V \setminus \{s,t\}]$ *adjacent to both* s *and* t.
Indeed, if there is none, then the graph G is not connected.

Suppose that there exist two connected components denoted by $G[C_i]$ and $G[C_j]$, such that

$$s,t \in N(C_i) \cap N(C_j).$$

Since S is a connected separator, we have $C_i \cap S \neq \phi$ and $C_j \cap S \neq \phi$. Then, $G[S]$ is not connected, a contradiction.
We conclude that there is an unique component C_{i^*}, for some $i^* \in \{1,...,k\}$, such that $s,t \in N(C_{i^*})$.

- Let $U := V \setminus (\{s,t\} \cup C_{i^*})$.
  Then, we have necessarily $U \cap S = \phi$ since $G[S]$ is connected. Therefore, we have $x_v^S = 0$, for all $v \in U$. Thus, suppose that $G[V \setminus \{s,t\}]$ is connected.

Study of the dimension: Let us now study the dimension of the polyhedron $P_{st}(G)$.

Theorem 4.4.1 ***(dimension of polyhedron)***:
Let U_0, U_1, U_2 *and* U_3 *be a subset of* V *defined as follows:*

- $U_0 := N(s) \cap N(t)$, *the set of vertices adjacent to* s *and* t.
- U^* *the set of articulation vertices for which there exists at least one connected component* C *of* $G[V \setminus \{s,v,t\}]$ *such that there is a path between* s *and* t *in the graph* $G[V \setminus C]$. *That is,* $U^* := U_1 \cup U_2 \cup U_3$, *where* U_i *is one of the following three configurations (shown in figure 4.1):*
  - U_1 *the set of vertices of articulation* $v \in V \setminus \{s,t\}$ *of* $G[V \setminus \{s,t\}]$ *such that there exists at least two related components* C_1 *and* C_2 *in* $G[V \setminus \{s,t,v\}]$ *each connected with the vertices* s *and* t *in* G:

$$s \in N(C_i) \text{ and } t \in N(C_i), \forall i = 1,2.$$

– *U_2 the set of vertices of articulation $v \in V \setminus \{s,t\}$ of $G[V \setminus \{s,t\}]$ such that there exists a unique connected component C in $G[V \setminus \{s,t,v\}]$ such that:*

$$s,t \in N(C_1),$$

and there are at least two connected components C_2 and C_3 in $G[V \setminus \{s,t,v\}]$

$$s \in N(C_2), s \notin N(C_3), t \notin N(C_2), t \in N(C_3).$$

– *U_3 the set of vertices of articulation $v \in V \setminus \{s,t\}$ of $G[V \setminus \{s,t\}]$ such that there exists two connected components C_1 and C_2 in $G[V \setminus \{s,t,v\}]$ such that :*

$$s \in N(C_i) \text{ and } t \notin N(C_i), \forall i = 1,2,$$

and there are two connected components C_3 and C_4 in $G[V \setminus \{s,t,v\}]$ such that:

$$s \notin N(C_i) \text{ and } t \in N(C_i), \forall i = 3,4,$$

and there is no connected component C of $G[V \setminus \{s,t\}]$ such that $s,t \in N(C)$

Then, we have dim $(P_{st}(G)) = |V| - 2 - |U_0 \cup U_1 \cup U_2 \cup U_3|$.

Proof. Let $x \in P_{st}(G)$. Trivially, we have dim $(P_{st}(G)) \leq |V| - 2$ since $x_s = x_t = 0$. Let us show that $(P_{st}(G)) = |V| - 2 - |U_0 \cup U_1 \cup U_2 \cup U_3|$. There are three cases:

- Let $v \in U_0$. Then, v is adjacent to s and t. Hence, it is necessary that v satisfies $v \in S$. Therefore, $x_v = 1$.
- Let $v \in V \setminus \{s,t\}$ such that $v \notin N(s)$ or $v \notin N(t)$.
  - Suppose that $v \in U_1$. Thus, there are two connected components C_1 and C_2 as

    $$s \in N(C_1)\, t \in N(C_1)\, s \in N(C_2)\, t \in N(C_2).$$

    Then, since $C_1 \cap S \neq \phi$ (because there exists a chain of s to t unique compound of vertices of C_1) and $C_2 \cap S \neq \phi$ (because there exists a chain of s to t unique compound of vertices of C_2), to guarantee the connectivity of $G[S]$, one must have $x_v = 1$.

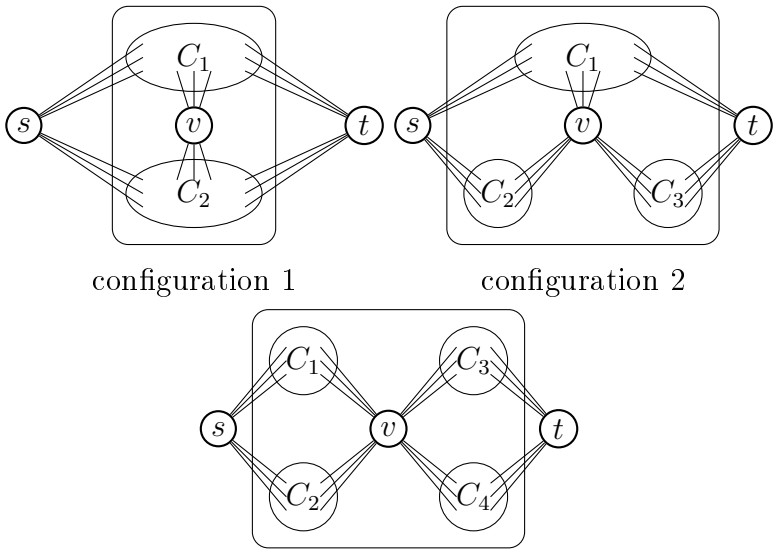


Figure 4.1 – Special articulation assembly configurations v

– Suppose that $v \in U_2$. Thus, there exists a unique connected component C_1 such as

$$s, t \in N(C_1),$$

and there are two connected components C_2 and C_3 as

$$s \in N(C_2), t \notin N(C_2), s \notin N(C_3), t \in N(C_3).$$

Then, since $C_1 \cap S \neq \phi$ (because there exists a chain of s to t composed of unique vertices of C_1) and $(C_2 \cup \{v\} \cup C_3) \cap S \neq \phi$ (since there exists a unique $s-t$-chain of vertices of $C_2 \cup \{v\} \cup C_3$), to guarantee the connectivity of $G[S]$, one must have $x_v = 1$.

– Suppose that $v \in U_3$. Thus, there are two connected components C_1 and C_2 as

$$s \in N(C_1), t \notin N(C_1), s \in N(C_2), t \notin N(C_2).$$

And, there are two connected components C_3 and C_4 as

$$s \notin N(C_3), t \in N(C_3), s \notin N(C_4), t \in N(C_4).$$

Then, as $(C_1 \cup C_2) \cap S \neq \phi$ and $(C_3 \cup C_4) \cap S \neq \phi$, to guarantee the connectivity of $G[S]$, one must have $x_v = 1$.

Thus, we have $\dim(P_{st}(G)) \leq |V| - 2 - |U_0 \cup U_1 \cup U_2 \cup U_3|$. Let us then show that there are $|V| - 1 - |U_0 \cup U_1 \cup U_2 \cup U_3|$ affine independent points in the polyhedron $P_{st}(G)$. Consider the subsets $V \setminus \{s,t\}$ and $V \setminus \{s,v,t\}$ for all $v \in V \setminus (\{s,t\} \cup U_1 \cup U_2 \cup U_3)$. These subsets are st-connected separators and their incidence vectors are trivially affine independent. Thus, we have $\dim(P_{st}(G)) = |V| - 2 - |U_0 \cup U_1 \cup U_2 \cup U_3|$.⊡

4.4.1 Articulation inequalities

Let $U \subseteq V \setminus \{s,t\}$ such that the graph $G[V \setminus (\{s,t\} \cup U)]$ is not connected and $G[U]$ is connected. Thus, the set U is a connected articulation set for the graph $G[V \setminus \{s,t\}]$. Let $C_1, ..., C_k$ be the subsets of V corresponding to the k connected components $G[C_1], ..., G[C_k]$ of $G[V \setminus (\{s,t\} \cup U)]$. Finally, to avoid obtaining redundant inequalities with constraint (4.3.2), we will assume that $s \notin N(U)$ or $t \notin N(U)$ and $I = \{1, ..., k\}$.

Under these assumptions, we have the following valid inequality:

$$x(U \cap (N(v) \cup N(w))) \geq x_v + x_w - 1, \tag{4.4.1}$$

where v and w are two vertices derived from two distinct components of $G[V \setminus (\{s,t\} \cup U)]$. Thus, if each vertex $u \in U$ is adjacent to either v or w, we obtain the following inequality:

$$x(U) \geq x_v + x_w - 1. \tag{4.4.2}$$

If we suppose that for all $i \in \{1, ..., k\}$, there is a path between s and t in the graph $G[V \setminus C_i]$, then, the following inequality is valid:

$$x(U) \geq 1 \tag{4.4.3}$$

This inequality is valid if one of the following three configurations (represented by figure (4.2)) is verified:

(C1) There is subset I^* of I such that $|I| \geq 2$ and, for all $i \in I^*$, we have

$$s, t \in N(C_i).$$

(C2) There is a unique $i^* \in I$ such that:

$$s, t \in N(C_{i^*}).$$

And, there are two disjoints subsets I_1 and I_2 of I, such that

- $s \in N(C_i)$ *and* $t \notin N(C_i)$ for all $i \in I_1$, and
- $s \notin N(C_i)$ *and* $t \in N(C_i)$ for all $i \in I_2$

(C3) For all $i \in I$, we have $s \notin N(C_i)$ or $t \notin N(C_i)$. And there are two disjoints subsets I_1 and I_2 of I, such that $\mid I_1 \mid \geq 2$, $\mid I_2 \mid \geq 2$ and

- $s \in N(C_i)$ *and* $t \notin N(C_i)$ for all $i \in I_1$, and
- $s \notin N(C_i)$ *and* $t \in N(C_i)$ for all $i \in I_2$

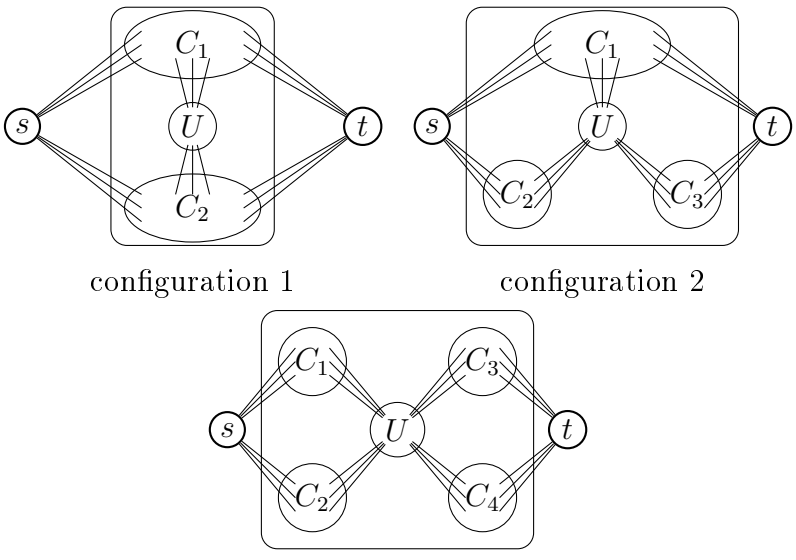


Figure 4.2 – Special articulation assembly configurations U

Case of the first two configurations: in the case of the first two configurations, we can improve the previous inequality (4.4.3). If I^* refers to the maximal subsets of I such that, for any $i \in I^*$, we have

$$s, t \in N(C_i),$$

Hence, we observe that for all $i \in I^*$, we have necessarily $x(C_i) \geq 1$. Thus, the following inequality is obtained:

$$\sum_{u \in U} k(u).x_u \geq |I^*|, \tag{4.4.4}$$

where $k(u)$ is the number of the connected components C_i, with $i \in I^*$, such that, there is an edge between u and some vertex of C_i.
Note that if each vertex of U is adjacent to exactly one connected component C_i, $i \in I^*$, (i.e. $k(u) = 1$), we obtain the following valid inequality:

$$x(U) \geq |I^*|. \tag{4.4.5}$$

Let $u \in U$, by definition of $k(u)$, if $u \notin \cup_{i \in I_{i^*}} N(C_i)$, then, $x(u) = 0$. Thus, inequality (4.4.4) can be written as follows:

$$\sum_{u \in (U \cap (\cup_{i \in I^*} N(C_i)))} k(u).x_u \geq |I^*|, \tag{4.4.6}$$

we have
$\sum_{u \in (U \cap (\cup_{i \in I^*} N(C_i)))} k^*.x_u \geq \sum_{u \in (U \cap (\cup_{i \in I^*} N(C_i)))} k(u).x_u \geq |I^*|$
where $k^* := \max_{u \in U} k(u)$.
Thus, the following inequality is valid

$$x\left(U \cap \left(\bigcup_{i \in I^*} N(C_i)\right)\right) \geq \left\lceil \frac{|I^*|}{k^*} \right\rceil, \tag{4.4.7}$$

4.4.2 Partition inequalities

Let $U := \{v_1, ..., v_p\}$ a set of p vertices and $(V_1, ..., V_p)$ a partition of the set of vertices such that $v_i \in V_i$, For all $i \in [1; p]$. The following inequality is valid:

$$y(\delta(V_1, ..., V_p)) \geq x(U) - 1. \tag{4.4.8}$$

This inequality is not separable in polynomial time even by fixing U.

4.5 Heuristic

In this section, we introduce an algorithm to find a set of connected separator in a short time. This algorithm depends on two stages. the first stage is to find a set of separator S by solving the problem (P1) (in the next subsection) integer programming. The second stage is to check the solution S, if it is connected then is done. Otherwise, we try to find vertices (one or more) that makes S connected.
Let $C_1, ..., C_k$ be the connected components of subgraph $G(S)$, and let $u \notin S \cup \{s, t\}$, we define C_i^u for $i = 1, ..., k$ as following: $C_i^u = 1$ if $(uv) \in E$ for some vertex $v \in C_i$, $C_i^u = 0$ in the otherwise.
We provide details for our algorithm for CVSP.

Algorithm 3 algorithm 3

1: *Solve P1 to find S.*
2: **if** *S is connected* **then STOP**
3: *Find connected components of S* $(C_1, ..., C_k)$.
4: *Find* $u^* \notin S \cup \{s, t\}$ *such that* $\sum_{i=1}^{k} C_i^{u^*}$ *is maximum,* $S := S \cup \{u^*\}$
5: **goto** *2*

4.5.1 Computational experiments

Our algorithm was programmed in C++ and compiled with GNU g++ under GNU/Windows7 running on laptop Dell has an Intel Core i7-4800MQ with 2.70 GHz and 8 GB of RAM. The 30 instances in the first class, called DIMACS graphs, come from the DIMACS challenge on graph coloring. The graphs included have (95 to 450)

vertices. These instances can be download from HTTP://www.mat.gisa.cmu.edu/COLOR/instances.html.
In the second and third class, set of 45 more challenging graphs from the well-known Market Matrix repository.These instances can be download from HTTP://math.nist.gov/MatrixMarket. The fourth class, set of 16 graphs come from Mixed Integer Problem Library. These instances can be downloaded from HTTP://miplib.zib.de.

For our experiments on these graphs, all the vertices are given unit weights. They have been conducted in two steps. We first calculate maximum number of disjoint paths between s and t called α_{st}, which is represents the lower bound of any cardinality of any st-separator. We used the code developed by Cherkassky and Goldberg [18] for the maximum flow problem corresponding to calculate α_{st}(Source code available at [77]). We then solved the following mixed-integer program (we call it $P1$) using Ilog-CPLEX 12.6 [78] to find minimum st-separator. The above steps are represent the first step from our algorithm.

$$\text{Min} \sum_{v \in V} (x_v)$$

subject to

$$x_s = 0, \quad x_t = 0 \tag{4.5.1}$$

$$x_u + x_v \leq 1 \quad , \quad x_v + x_u \leq 1 \qquad \forall (uv) \in E \tag{4.5.2}$$

$$x_v + x_u \leq 1 \qquad \forall v, u \in V \tag{4.5.3}$$

$$\sum_{v \in V} (x_v + x_u) = n - \alpha_{st} \tag{4.5.4}$$

$$x_v, x_u \in \{0, 1\} \qquad \forall v, u \in V \tag{4.5.5}$$

The results of our computational experiments are presented in tables 1,2,3 and 4, where each line corresponds to one specific instance. In the tables, there are nine columns. The first column represents the name of instance. The second is the number of vertices of graph (order of graph). The third is the source s. The fourth is the sink t. The fifth is the minimum st-separator S that we find it by solving the problem integrally. The sixth shows the status of column five; if the vertices in set S

connected the status *yes* else *no*. The seventh represents the number of components in set S, note that if S connected then the number of components is 1. The eighth refers to the numbers of verices in S that make S connected, and the final column represents the total time in seconds for solving the problem by our algorithm.

Table 4.1 – DIMACS instances (30 instances)

Instances	n	s	t	*min.separator*	*connected*	*no. of components*	*connected separator*	*time*
DSJC125.1	125	59	11	6	*no*	3	8	1.3
DSJC125.5	125	15	115	58	*yes*	1	58	1.06
DSJC125.9	125	41	1	107	*yes*	1	107	0.83
games120	120	41	22	10	*no*	2	11	0.27
miles500	128	68	84	2	*yes*	1	2	0.20
miles750	128	17	111	9	*yes*	1	9	0.33
miles1000	128	17	119	11	*yes*	1	11	0.20
myciel6	95	4	86	6	*yes*	1	6	0.11
myciel7	191	180	64	8	*no*	8	14	0.77
queen10-10	100	30	53	27	*yes*	1	27	0.22
queen11-11	121	80	41	34	*yes*	1	34	0.37
queen12-12	144	3	41	33	*yes*	1	33	0.12
queen13-13	169	25	41	36	*yes*	1	36	0.49
queen14-14	196	13	137	39	*yes*	1	39	0.25
queen15-15	225	7	41	42	*yes*	1	42	0.29
queen16-16	256	32	107	45	*yes*	1	45	0.18
le450-5a	450	111	41	24	*no*	2	25	0.40
le450-5b	450	200	41	23	*no*	3	24	1.16
le450-5c	450	77	312	31	*yes*	1	31	0.33
le450-5d	450	322	3	37	*yes*	1	37	0.35
le450-15b	450	102	400	7	*yes*	1	7	0.32
le450-15c	450	55	112	59	*yes*	1	59	0.52
le450-15d	450	9	87	84	*yes*	1	84	0.40
le450-25a	450	88	399	29	*no*	3	30	1.01
le450-25b	450	400	222	18	*no*	3	20	2.40
le450-25c	450	12	288	58	*yes*	1	58	0.47
le450-25d	450	59	411	50	*yes*	1	50	0.15
anna	138	11	100	2	*yes*	1	2	0.29
flat300-28-1	300	114	280	135	*no*	12	140	2.44
DSJC250.1	250	8	93	23	*no*	3	25	1.33

Table 4.2 – MM-II instances (20 instances)

Instances	n	s	t	*min.separator*	*connected*	*no. of components*	*connected separator*	*time*
L125.ash608	125	14	82	6	*yes*	1	6	0.3
L125.will199	125	11	53	2	*yes*	1	2	0.3
L125.west0167	125	77	2	2	*yes*	1	2	0.21
ash331	125	15	91	3	*yes*	1	3	0.3
west0132	132	6	101	2	*yes*	1	2	0.3
rw136	136	12	133	5	*yes*	1	5	0.3
bcspwr03	118	95	117	7	*yes*	1	7	0.22
gre-115	115	54	101	8	*yes*	1	8	0.3
L125.dwt-162	125	33	105	6	*yes*	1	6	0.37
L125.can-187	125	22	112	14	*yes*	1	14	0.28
L125.gre-185	125	56	112	13	*yes*	1	13	0.3
L125.can-161	125	97	78	17	*yes*	1	17	0.21
L125.lop163	125	61	95	16	*yes*	1	16	0.23
can-144	144	82	1	18	*yes*	1	18	0.22
lund-a	147	11	136	25	*yes*	1	25	0.31
L125.bcsstk05	125	9	93	24	*yes*	1	24	0.43
L125.dwt-193	125	43	46	24	*yes*	1	24	0.33
L125.fs-183-1	125	73	19	36	*yes*	1	36	0.35
bcsstk04	132	98	36	48	*yes*	1	48	0.33
arc130	130	36	18	32	*yes*	1	32	0.36

Table 4.3 – MM-HD instances (25 instances)

Instances	n	s	t	*min.separator*	*connected*	*no. of components*	*connected separator*	*time*
L100.steam2	100	49	16	28	*yes*	1	28	0.36
L100.cavity01	100	24	99	26	*yes*	1	26	0.39
L100.fidap021	100	3	71	16	*yes*	1	16	0.31
L100.fidap025	100	44	27	18	*yes*	1	18	0.36
L100.wm1	100	56	17	38	*yes*	1	38	0.38
L100.wm2	100	82	50	45	*yes*	1	45	0.39
L100.wm3	100	61	55	30	*yes*	1	30	0.38
L100.fidapm02	100	34	96	40	*yes*	1	40	0.41
L100.e05r0000	100	30	49	30	*yes*	1	30	0.35
L100.fidap001	100	1	17	39	*yes*	1	39	0.26
L100.rbs480a	100	91	89	30	*yes*	1	30	0.35
L100.fiap022	100	18	11	53	*yes*	1	53	0.31
L100.fidap027	100	10	44	34	*yes*	1	34	0.33
L100.fidap002	100	2	31	52	*yes*	1	52	0.32
L120.fidap025	120	93	3	18	*yes*	1	18	0.33
L120.fidap021	120	69	58	24	*yes*	1	24	0.35
L120.cavity01	120	6	3	21	*yes*	1	21	0.42
L120.rbs480a	120	71	48	33	*yes*	1	33	0.38
L120.wm2	120	55	3	29	*yes*	1	29	0.38
L120.e05r0000	120	4	23	42	*yes*	1	42	0.33
L120.fidap022	120	18	11	54	*yes*	1	54	0.43
L120.fidap001	120	36	93	52	*yes*	1	52	0.35
L120.fidapm02	120	2	22	51	*yes*	1	51	0.35
L120.fidap002	120	0	33	57	*yes*	1	57	0.31
L120.fidap027	120	8	44	37	*yes*	1	37	0.46

Table 4.4 – miplib instances (15 instances)

Instances	n	s	t	*min.separator*	*connected*	*no. of components*	*connected separator*	*time*
air03.p	124	28	121	39	*yes*	1	39	0.33
bell3a.p	127	52	2	3	*no*	2	4	0.86
bell4.p	101	69	27	4	*yes*	1	4	0.36
blend2.p	169	7	129	10	*yes*	1	10	0.33
khb05250.p	100	67	30	25	*yes*	1	25	0.32
l152lav.p	97	90	10	17	*yes*	1	17	0.46
misc03.p	96	72	68	49	*yes*	1	49	0.45
misc07.p	212	10	165	32	*yes*	1	32	0.53
mod010.p	146	33	51	47	*yes*	1	47	0.49
noswot.p	182	155	139	11	*yes*	1	11	0.39
p0201.p	113	81	24	19	*yes*	1	19	0.33
p0808a.p	136	16	112	2	*no*	2	17	1.47
p0808aCUTS	246	4	139	6	*no*	4	8	1.24
stein27-r.p	118	49	9	37	*yes*	1	37	0.36
vpm1.p	234	222	183	2	*yes*	1	2	0.35

We note that in table (4.1), the separator of the 30% of instances are not connected, and we add (0.2-3.2)% from rest of the vertices to make the separator is connected.
In tables (4.2) and (4.3), all separators in all instances are connected.
In table (4.4), the separator of the 20% of instances are not connected, and we add (0.8-11)% from rest of the vertices to make the separator is connected.

4.6 Conclusions and Remarks

In this chapter, we have considered three formulations of *st*-connected separator problem. Also, we gave some types of valid inequalities for the associated polytope. Also, we gave a heuristic to solve this problem in a short time. To develop this work, we need to solve this problem exactly by using Branch and Cut. Also, we need to expand our search within this problem by studying particular cases of graphs such as trees, wheel graphs, Halin graphs, ...etc.

Conclutions

In this thesis, we have studied the vertex separator problem. In the first two chapters, we gave some basic notations concering combinatorial optimization and partitioning graph like complexity, exact and heuristic methods,... etc..

In the third chapter, we have studied the vertex separator problem (VSP) and gave an integer programming formulation for this problem. Moreover, we gave two efficient heuristics based on neighborhood searching. Which use some valid inequalities depend on maximum number of node-disjoint paths between any two vertices α_{uv} and the constant β. This has been implemented to solve DIMACS and others instances. In the experiment results, we compared our results with [16, 10] and we gate results better in many of these instances, also we attaind an optimal solution for many. Future works include study of relaxations and use these heuristics and other inequalities as cutting planes by using Branch and Cut for find exact solution for the VSP.

In the last chapter of the thesis,we have studied the *st*-connected vertex separator (*st*-CVS) problem, we have proposed three integer programming formulations for this problem. Also we investigated the related polyhedron and disscussed its polyhedral structure, and gave two type of valid inequalities articulation and partition inequalities. Moreover, we proposed a heuristic to solve this problem.
This heuristic solved larg of instances, (DIMACS, Matrix Market, MIPLIP) instances, quickly but we do not have any previous information to compar with it. Therefor, these results can serve as references for future *st*-CVS problem.
Future work include more efficient *st*-connected vertex heuristics and more valid inequalities so as to use these results as cuts when we use Branch and Cuts to find exact solutions. Also, it would be interesting to characterize particular cases of graphs like trees, wheel graphs, Halin graphs,...etc.

Chapter 5

Compte-rendu

Optimisation Combinatoire

L'optimisation combinatoire (ou discrète) est l'un des domaines les plus actifs dans l'interface de la recherche opérationnelle, l'informatique et les mathématiques appliquées.
La cible de l'étude de l'optimisation combinatoire est de trouver la fonction maximiser (ou minimiser) de nombreuses variables soumises à des contraintes. La caractéristique distinctive de l'optimisation combinatoire est que certaines des variables doivent appartenir à un ensemble discret, typiquement un sous-ensemble d'entiers.

En général, les problèmes relatifs à l'optimisation combinatoire peuvent être formulés comme suit. Soit E=$\{e_1, e_2, ..., e_n\}$ soit un ensemble fini appelé basic set . Soit Ω une famille de sous-ensembles de E. Si π $in\Omega$, alors $c(\pi) = \sum_{e_i \ in \ pi} c(e_i)$ représente le poids de π. Le problème est de déterminer un élément de Ω avec les poids plus grands (ou plus petits). Dans le problème d'optimisation combinatoire, l'ensemble Ω est appelé l'ensemble des solutions du problème. En d'autres termes,

$$\min(\text{ou } \max)\{c(\pi) : \pi \in \Omega\}$$

Le terme *combinatorial* fait référence à la structure discrète de *Omega*. En général, cette structure est représentée par un graphique. Le terme *optimisation* signifie que nous cherchons le meilleur élément dans l'ensemble des solutions réalisables.

En général, cet ensemble contient un nombre exponentiel de solutions ; par conséquent, nous ne pouvons pas nous attendre à résoudre un problème d'optimisation combinatoire en énumérant exhaustivement toutes ses solutions. Un tel problème est alors considéré comme un problème difficile. Le défi consiste à développer des algorithmes qui sont prouvés ou pratiquement meilleurs que d'énumérer toutes les solutions possibles.

Les problèmes d'optimisation combinatoire se posent dans diverses applications, y compris la conception de réseaux de communications, la conception VLSI, la vision industrielle, la planification d'équipe aérienne, la planification d'entreprise, la conception assistée par ordinateur et la fabrication. Outre les applications, l'optimisation discrète a des aspects qui le relient à d'autres domaines des mathématiques (par exemple, l'algèbre, l'analyse et l'optimisation continue, la géométrie, la logique, l'analyse numérique, la topologie et bien sûr d'autres sous-disciplines de mathématiques discrètes telles que Linear et Integer Programmation, théorie des graphes, intelligence artificielle et théorie des nombres). Tous ces problèmes, lorsqu'ils sont formulés mathématiquement comme la minimisation ou la maximisation d'une certaine fonction définie sur un domaine donné, ont une communauté de discrétion.

En fait, l'optimisation combinatoire est étroitement liée à la théorie de l'algorithme et à la théorie de la complexité computationnelle. La section suivante présente les problèmes de calcul de l'optimisation combinatoire.

Complexité informatique

Computational Complexity Theory est une branche de la théorie du calcul en informatique théorique et en mathématiques qui se concentre sur la classification des problèmes de calcul en fonction de leur difficulté inhérente et la relation entre ces classes. Cette théorie est née tôt en 1844 par Gabriel Lame, mais le champ a vraiment commencé à prospérer en 1962 par Edmonds [30] et Cook en 1971 [19] qu'ils ont offert un cadre pour classer les problèmes selon leur difficulté. En 1972, Karp [48] introduisit plus d'informations sur 21 problèmes combinatoires et graphiques théoriques, tous infâmes pour leur insolation computationnelle, qui sont NP-complets dans la théorie de la complexité.

A *problem* est une question générale à laquelle nous souhaitons trouver une réponse. Cette question a généralement des paramètres ou des variables dont les valeurs doivent encore être déterminées. Un problème est posé en donnant une liste de ces paramètres ainsi que les propriétés auxquelles la réponse doit se conformer.
Une *instance* d'un problème est obtenue en donnant des valeurs explicites à chacun des paramètres du problème instancié.
Un *algorithme* est une suite d'opérations élémentaires qui, lorsqu'elles sont données comme une instance d'un problème en entrée, donnent la solution de ce problème en sortie après l'exécution de l'opération finale.

A *decision problem* est une question concernant l'existence, pour une instance donnée, d'une configuration telle que cette configuration elle-même ou sa valeur soit conforme à certaines propriétés. La solution à un problème de décision est une réponse à la question associée au problème. En d'autres termes, cette solution peut être *yes*, une telle solution existe ou *no*, une telle solution n'existe pas.
Un *algorithm* est une procédure computationnelle bien définie qui prend certaines valeurs ou un ensemble de valeurs en entrée et produit une valeur ou un ensemble de valeurs en sortie. Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie. Le nombre de paramètres nécessaires à l'entrée pour décrire une instance d'un problème appelé *size* of problem.

L'efficacité d'un algorithme, évaluée comme la quantité de ressources informatiques dont il a besoin, détermine si un algorithme est bon ou non. Dans l'efficacité d'un algorithme, il y a deux ressources principales : (a) le temps (c'est-à-dire le nombre d'étapes de calcul élémentaires (combien de temps le meilleur algorithme nécessite pour résoudre le problème)) ; (b) l'espace (c'est-à-dire l'ampleur de la mémoire interne requise).

La complexité de calcul d'un problème peut être liée à l'algorithme qui le résout. De plus, n'importe quel problème peut être classé dans certaines classes de complexité en fonction de sa complexité. Les classes de complexité les plus connues $textbfP - class$ (*polynomial*) et $textbfNP - class$ (*Aucun-déterministe Polynomial*) sont des collections de problèmes de décision. La classe **P** est une classe de complexité représentant l'ensemble de tous les problèmes de décision qui peuvent être résolus en temps polynomial. C'est-à-dire, étant donné un exemple du problème, la réponse "oui" ou "non" peut être décidée en temps polynomial.

La classe **NP** est une classe de complexité représentant l'ensemble de tous les problèmes de décision pour lesquels les instances où la réponse est "oui" ont des preuves qui peuvent être vérifiées en temps polynomial. Cela signifie que si une instance du problème et un certificat (parfois appelé un témoin) à la réponse étant oui, nous pouvons vérifier qu'il est correct en temps polynomial. On peut voir que la classe **P** appartient à **NP**. La question ouverte est de savoir si les problèmes textbf NP ont ou non des solutions de temps polynomiales déterministes.
La classe $\mathbf{NP} - Complete$ est une classe de complexité représentant l'ensemble des problèmes X dans **NP** pour lequel il est possible de réduire tout autre problème **NP** Y to X en temps polynomial.
Intuitivement, cela signifie que nous pouvons résoudre rapidement Y si nous savons comment résoudre rapidement X. Précisément, Y est réductible à X, s'il y a un temps polynomial f pour transformer des instances y de Y en instances $x = f(y)$ de X en temps polynomial avec propriété que la réponse à y est *yes* si et seulement si la réponse à $f(y)$ est oui.
La classe **NP-hard** est qu'un probl'eme X est **NP-hard**, s'il existe un probl'eme $\mathbf{NP} - Complete\ Y$, tel que Y soit réductible à X en temps polynomial. Mais puisque tout problème NP-Complete peut être réduit à tout autre problème NP-Complete en temps polynomial, tous les problèmes NP-Complete peuvent être réduits à n'importe quel problème NP-difficile en temps polynomial. Ensuite, s'il existe une solution à un problème NP-difficile en temps polynomial, il existe une solution à tous les problèmes NP en temps polynomial.
Nous notons que la plupart des problèmes d'optimisation combinatoire sont textbf NP-hard. L'une des approches les plus efficaces développées pour résoudre ces problèmes est l'approche dite polyédrique.

Éléments de la théorie polyédrique

Dans cette section, nous donnons quelques notions fondamentales de la théorie polyédrique sur l'optimisation linéaire et entière. D'autres références sont Schrijver cite S03, Nemhauser et Wolsey cite NAW88, et Mahjoub cite M13.

La théorie polyédrique traite des ensembles faisables de problèmes de programmation linéaire, appelés *polyhedra*. Maintenant, la théorie polyédrique peut être

considérée comme faisant partie de l'analyse convexe, qui est la branche des mathématiques qui étudie les ensembles convexes, c'est-à-dire les ensembles contenant les segments de droite entre chaque paire de points. Il existe deux façons de définir les polyèdres et les polytopes. Certains textes commencent par la définition d'un polytope comme une combinaison convexe d'un nombre fini de points. D'autres, commencent par la définition de polyhera via l'intersection de un nombre fini de demi-espaces. Les polyèdres sont apparus au début de l'histoire (comme les pyramides, les dés et autres signes de civilisation) et en géométrie (comme Platon, Euclide, Archimède). Cependant, tous étaient des polyèdres individuels et particuliers. Même au milieu du 18ème siècle, Euler a discuté son fameux théorème *(V-E + F = 2)* sans préciser quels sont les polyèdres auxquels il s'applique. Pendant le 19^{th} siècle, le centre d'attention concernant les polyèdres est passé à des convexes (plus ou moins explicitement déclarés).

Nous rappellerons d'abord quelques définitions et propriétés liées à la théorie polyédrique.

Donné n être un entier positif et x $dans\mathbb{R}^n$. On dit que x est une combinaison *linar* de $x_1, x_2, ..., x_n \in \mathbb{R}^n$ s'il existe m scaler $\delta_1, \delta_2 ..., \delta_m$ tel que $x = \sum_{i\in 1}^{m} \delta_i x_i$. Si $\sum_{i\in 1}^{m} \delta_i = 1$, alors x est dit *combinaison affine* de $x_1, x_2, ..., x_m$. si $\delta_i \geq 0$, pour tout $i \in \{1, ..., m\}$, nous disons que x est une combinaison *conic* de $x_1, x_2, ..., x_m$. si x est *affine* et *conic*, nous disons que x est *combinaison convexe*.
Le problème d'optimisation linéaire concerne la maximisation ou la minimisation d'une fonction linéaire $c^T x$ sur un polyèdre P a la forme

$$\max\{c^T x : x \in P\}.$$

Le polyèdre P est appelé la région réalisable, et tout vecteur dans P est une solution réalisable. Si la région faisable est non vide, le problème est appelé *faisable* et *infaisable* sinon. La fonction linéaire $c^T x$ est appelée *fonction objective*.
Si on sait que $P = \{x \in \mathbb{R}^n : Ax \leq b\}$, le problème d'optimisation a aussi une forme équivalente comme ci-dessous.

$$max\{c^T x : Ax \leq b, x \geq 0\}. \tag{5.0.1}$$

Aucun algorithme polynomial-temps n'est connu pour résoudre un problème de programmation linéaire d'entier en général. En fait, le problème de programmation

linéaire entier entier est NP-complet.
Un polyèdre P est appelé un *polyèdre entier* s'il est l'enveloppe convexe des vecteurs entiers contenus dans P. Ceci est équivalent à P est rationnel et chaque face de P contient un vecteur entier. Donc un polytope P est entier si et seulement si chaque sommet de P est entier. Si un polyèdre Pest un entier, alors le probl'eme de programmation linéaire $max\{c^T x : Ax\ leqb\}$ a une solution optimale entière si elle est finie. Par conséquent, dans ce cas,

$$max\{c^T x : Ax \leq b, x \in \mathbb{Z}^n\} = max\{c^T x : Ax \leq b\}.$$

Partitionnement de graphe

Graphs sont des structures formées par un ensemble de sommets (également appelés *noeuds*) et un ensemble d'arêtes qui sont des connexions entre des paires de sommets, tandis que le mot *partition* représente l'action de création d'une partition en segmentant un ensemble en plusieurs parties. En mathématiques, une partition d'un ensemble X est une famille de parties disjointes de X dont l'union est l'ensemble X. Par conséquent, chaque élément de X appartient à un, et seulement une des parties de la partition.
La création d'une partition consiste à distribuer un ensemble d'objets dans plusieurs sous-ensembles. Afin de distribuer les objets dans ces différents sous-ensembles, il peut être utile de les comparer.

Le partitionnement graphique est une discipline située entre l'informatique et les mathématiques appliquées, et *le graphe partition* est défini sur des données représentées sous la forme d'un graphe $G = (V, E)$, avec V sommets et E arêtes, de sorte qu'il est possible de partitionner G en composants plus petits avec des propriétés spécifiques.
Une implication importante du théorème de séparateur est que tout graphe avec un mineur exclu fixe avec un degré maximum limité par d peut être partitionné en petits composants de taille au plus poly $(d, 1/\varepsilon)$ en supprimant seulement un ε -fraction des arêtes.

Il existe de nombreuses applications de partitionnement de graphes dans différents domaines, comme la conception de logiciels et de matériel (par exemple, Traitement parallèle [12] et Conception physique de circuits numériques pour l'intégration à très grande échelle (VLSI) [50, 20], réseaux et conception routière [52, 51] (Par exemple, les arêtes pourraient être des segments de route et des intersections de nœuds.), Image Processing [67, 15] (partitionner les pixels d'une image en groupes

correspondant à des objets.), biologie [46, 62] (services d'information géographique et cartographie physique de l'ADN (Deoxyribo Nucleic Acid, constituant basique du gène)). Par exemple, dans la conception VLSI, il est souvent nécessaire de diviser les conceptions trop grandes pour qu'elles correspondent à une seule puce dans de plus petites parties de " à peu près " de même taille, où chaque partie est mappée sur une puce séparée. Par rapport aux connexions au sein d'une puce, les connexions entre les puces sont coûteuses, lentes et consommatrices d'énergie. Par conséquent, il est souhaitable de réduire la capacité de la coupe définie par une partition.

Description formelle du problème de partitionnement du graphe

Soit $G = (V, E)$ un graphe de poids de nœud $f : V \longrightarrow \mathbb{R}$, $n =\mid V \mid$ et $m =\mid E \mid$ et $P_k = \{V_1, ..., V_k\}$ un ensemble de k sous-ensembles de V. P_k est dite partition de G si :

- aucun élément de P_k n'est vide :
  $\forall i \in \{1, ..., k\} \neq \phi$.
- les éléments de P_k sont disjoints par paires :
  $\forall (i, j) \in \{1, ..., k\}^2, i \neq j, V_i \cap V_j = \phi$;
- l'union de tous les éléments de P_k est égale à V :
  $\cup_{i=1}^{k} V_i = V$

Les éléments V_i de P_k sont appelés *parties* de la partition. Le nombre k est appelé le *cardinality* de la partition, ou le nombre de parties de la partition.
Une contrainte *balance* exige que tous les blocs (partitions) aient des poids égaux. Plus précisément, il exige que,$\forall i \in \{1, ..., k\} :\mid V_i \mid \leq L_{max} := (1 + \epsilon)\lceil \mid V \mid / k \rceil$ pour un certain paramètre de déséquilibre $\epsilon \in \mathbb{R}_{\geq 0}$. Un bloc V_i est surchargé si $\mid V_i \mid > L_{max}$. Le problème général de partitionnement des graphes consiste à trouver la partition P_k qui minimise f.

Méthodes de graphe de partition

Dans cette section, nous discutons des algorithmes et des méthodes qui fonctionnent avec le graphe entier et calculons une solution. Ces algorithmes sont souvent utilisés pour des graphes plus petits ou sont appliqués en tant que sous-programmes dans des méthodes plus complexes telles que la recherche locale ou les algorithmes multiniveaux.

Méthodes exactes

Toutes ces méthodes peuvent généralement résoudre seulement de très petits problèmes tout en ayant très de grands temps de fonctionnement ou s'ils peuvent résoudre de grandes instances de partitionnement en deux temps [23, 24], ils dépendent fortement de la largeur de bissection du graphe. Les méthodes qui résolvent le problème général de partitionnement de graphe [33, 71] ont des temps de parcours immenses pour des graphes avec quelques centaines de nœuds. De plus, l'évaluation expérimentale de ces méthodes ne prend en compte que les petits numéros de blocs $k \leq 4$.

Méthodes heuristiques

C'est une approche pour trouver des solutions dans un laps de temps raisonnable. Les méthodes traditionnelles ne fonctionnent pas bien sur un grand espace pour donner des résultats dans un laps de temps raisonnable. L'approche heuristique ne donne pas de garanties pour toujours trouver une solution optimale. Beaucoup de problèmes du monde réel comme NP-dur peuvent être résolus en utilisant une approche heuristique. Les résultats qui sont trouvés par de bonnes solutions approximatives sont souvent NP-dur. Malgré ces résultats négatifs, plusieurs méthodes pour trouver de bonnes solutions approximatives pour plusieurs classes restreintes de graphes ont été développées au fil des ans.

Familles de problèmes de partitionnement de graphe

Dans cette section, nous rappelons que les types de familles de problèmes de partitionnement de graphe dépendent de la suppression des sommets ou des arêtes d'un graphe afin de disjoindre les composants avec des propriétés spécifiques.

(-) **Le problème de coupe de vertex à** $k - way$

(-) **Le problème** k **-separator**

(-) **Le problème du séparateur de vertex**

(-) **Le problème du séparateur de sommets multi-terminaux**

(-) **Le problème de Multiway Cut**

(-) **Problème multicoupe Vertex**

(-) **Problème des sommets critiques (CNDP)**

(-) **Le problème de coupe de bord à** $k - way$

(-) **Le problème de partitionnement de graphe équilibré**

Problème Vertex Separator

Le séparateur de sommets d'un graphe est un sous-ensemble de sommets dont la suppression déconnecte le graphe en au moins deux composants connectés non-vides. Si on a $G = (V, E)$ un graphe non-orienté et un entier positif $\beta(n)$, où $n = |V|$, le problème du séparateur de sommets est de trouver une partition de V en trois classes non-vide A, B et C telles que :

(i) Il n'y a pas de limite entre A et B ;

(ii) $\max\{|A|, |B|\} \leq \beta(n)$;

(iii) $|C|$ est minimum.

Les sous-ensembles A et B sont appelés *shores* du séparateur C. Par commodité, une partition $\{A, B, C\}$ de V qui vérifie (i) et (ii) sera aussi appelée séparateur.
Le VSP est NP-dur [13, 37]. Même pour des classes spéciales de graphes structurés, VSP reste NP-dur, (si G est planer graph [36], ainsi que pour [39], graphes bipartites [38], graphes de grille et graphiques de disque d'unité [27]).

Lorsque $\beta(n) = n - k$ pour une constante positive k, le problème devient polynomialement soluble [5]. Comme cela a été mentionné dans [5], le VSP est trivial si $\beta(n) = 1$ et est polynomialement résoluble si $\beta(n) \geq n - 1$.

Le VSP est utile pour de nombreux algorithmes graphiques et possède un certain nombre d'applications : (VLSI) pour la division des circuits en sous-systèmes plus petits, avec un petit nombre de composants à la frontière entre les sous-systèmes. La version décisionnelle du VSP consiste à trouver une valeur de séparation de sommet supérieure à un seuil donné. Il a des applications sur la conception de compilateur de langage informatique et les algorithmes exponentiels. Dans la conception du com-

pilateur, le code à compiler peut être représenté sous la forme d'un graphe orienté acyclique (DAG) où les sommets représentent les valeurs d'entrée du code ainsi que les valeurs calculées par les opérations du code. (Réseaux de communication, bioinformatique, dessin et traitement du langage naturel (voir [5] pour une enquête sur les applications VSP.)).

Plusieurs algorithmes exacts ont été proposés pour résoudre VSP [5, 26, 28, 25, 16]. Ces algorithmes VSP sont capables de trouver des résultats optimaux dans un temps de calcul raisonnable (dans les 3 heures) pour des instances ayant jusqu'à 150 sommets, et peuvent échouer à résoudre des instances plus grandes, une variété d'algorithmes heuristiques ont été trouvés très efficace pour trouver des partitions quasi optimales. Benlic et Hao [10] et Zhang et Shao [75] ont utilisé la recherche locale BLS pour résoudre VSP. Jes 'us, Juan et Abraham [45] ont utilisé quatre algorithmes de shake et les ont incorporés dans un schéma de recherche de voisinage variable pour résoudre ce problème. Sanchez-Oro et al. [68] ont introduit plusieurs algorithmes VNS (Variable Neighbourhood Search), qui alternent entre une phase de recherche locale et une phase d'agitation. Hager et al. [40] ont proposé une approche d'optimisation continue.

Les méthodes heuristiques et méta-heuristiques, qui se sont révélées très utiles pour divers problèmes d'optimisation combinatoire NP-hard, n'ont pas été envisagées jusqu'à présent pour le VSP. Bien que ces méthodes n'aient pas de garantie formelle de performance, elles ont montré qu'elles étaient capables de fournir des solutions de qualité acceptable avec des efforts de calcul raisonnables même pour de très grands cas.

Les graphes que nous considérons sont finis, non orientés et connectés. Nous désignons un graphe par $G = (V, E)$, où V est l'ensemble des noeuds et E est l'ensemble des arêtes. Si e est un avec les noeuds finaux u et v, alors on écrit $e = (uv)$. Si $W \subseteq V$, le ensemble de bords ayant un nœud d'extrémité dans W et l'autre dans $\overline{W} = V \setminus W$ s'appelle *cut* et est noté par $\delta(W)$. L'ensemble des arêtes ayant les deux noeuds d'extrémité dans W sera noté $E(W)$. Si W_1, W_2 sont des sous-ensembles disjoints de V, alors $[W_1, W_2]$ dénote l'ensemble des les arêtes de G qui ont un nœud dans W_1 et l'autre dans W_2. Pour $U \subseteq V$ nous désignons par $G(U)$ le sous-graphe induit sur U (c'est-à- $G(U) = (U, E(U))$. Si $F \subseteq E$, alors $V(F)$ dénote l'ensemble des noeuds de F et $G(F)$ le sous-graphe de G induit par F.

Formulation

Soit $G = (V, E)$ un graphe non orienté et $\beta(n)$ un entier positif, où $n = |V|$. Pour tout séparateur $\{A, B, C\}$, nous associons un vecteur d'incidence $(x, y) \in \mathbb{R}^{2n}$ défini par $x_v = 1$ si $v \in A$ et 0 sinon, et $y_v = 1$ si $v \in B$ et 0 sinon. Le VSP est équivalent au programme linéaire entier :

$$max\{\sum_{v \in V}(x_v + y_v)$$

s.t.

$$x_u + y_v \leq 1 \qquad \forall (uv) \in E \qquad (5.0.2)$$

$$x_v + y_u \leq 1 \qquad \forall (uv) \in E \qquad (5.0.3)$$

$$x_v + y_v \leq 1 \qquad \forall v \in V \qquad (5.0.4)$$

$$1 \leq \sum_{v \in V} y_v \leq \beta(n) \qquad (5.0.5)$$

$$1 \leq \sum_{v \in V} x_v \leq \beta(n) \qquad (5.0.6)$$

$$x_v \in \{0, 1\} \qquad \forall v \in V \qquad (5.0.7)$$

$$y_v \geq 0 \qquad \forall v \in V \qquad (5.0.8)$$

Les premières et deuxièmes inégalités viennent du fait qu'il n'y a pas d'écart entre A et B.
Les troisièmes inégalités viennent du fait que $A \cap B = \emptyset$. Quatrième et cinquième inégalités viennent du fait que $A \neq \emptyset \neq B$ et $\max\{|A|, |B|\} \leq \beta(n)$.

Soit u et v être des sommets non adjacents. Notons par α_{uv} le nombre maximum de chemins disjoints entre u et v. Définir

$$\alpha_{min} = min\{\alpha_{uv} : u, v \in V, uv \in E\}.$$

Nous notons que pour tout chemin entre deux sommets $a \in A$ et $b \in B$, alors C a au moins un sommet en commun. Par conséquent $|C| \geq \alpha_{min}$. Ainsi, nous utilisons α_{min} comme limite inférieure de la cardinalité de tout séparateur.

Recherche de Neighborhood

Dans un graphe $G = (V, E)$, deux sommets u, v sont considérés comme voisins si uv est un front dans E. Le voisinage d'un sommet u, noté $N(u)$, est l'ensemble des sommets $v \in V$ tels que $uv \in E$. Le voisinage d'un sous-ensemble $A \subseteq V$, noté $N(A)$, est l'ensemble des sommets $v \in V \setminus A$ tel que $uv \in E$ pour $u \in A$.

Algorithme 2

Dans cet algorithme, nous choisissons le sommet ayant un degré minimum dans la première étape et le mettons dans l'ensemble A et trouvons le voisinage de celui-ci et les mettons dans l'ensemble C et le reste des sommets dans l'ensemble B. Dans la deuxième étape, choisir un sommet i pas dans A avec un voisinage minimum dans $B \setminus \{i\}$. Mettez i dans l'ensemble A, et ainsi de suite jusqu'à satisfaire la condition $|A| + |C| > n - \beta$.

Nous pouvons utiliser le résultat de cet algorithme comme borne inférieure lorsque nous résolvons l'intégralité du problème. Nous obtenons également un nouveau α_{min} appelé ($\alpha*_{min}$) supérieur ou égal à l'original α_{min}. nous fournissons des détails sur notre algorithme pour VSP.

Algorithm 4 algorithme 2

1: *choisissez le sommet a qui a un degré minimum (s'il y en a plus d'un, choisissez-en un).* $A = \{a\}; C = \phi; B = \phi$.
2: $C = N(A); B = V \setminus (C \cup A)$.
3: **while** $|A| + |C| \leq n - \beta$ **do**
4: *Soit* $i \in V \setminus A$ *tel que* $|N(i) \cap B|$ *est minimum* .
5: $A = A \cup \{i\}; C = N(A); B = V \setminus (A \cup C)$.
6: *End while*

Maintenant, appelons $(|A^*|, |B^*|, |C^*|)$ une solution de notre algorithme, nous essayerons de trouver une solution meilleure que $(|A^*|B^*|, |C^*|)$, autrement dit une solution $|A + B|$ telle que $|A + B| > |A^* + B^*|$, pour cela, nous ajouterons des inégalités comme coupures à notre problème. Les détails dans la section suivante.

Expériences de calcul

Ils ont été réalisés en deux étapes. Nous avons d'abord calculé α_{min}, la borne inférieure de toute cardinalité de tout séparateur. Nous avons utilisé le code développé par Cherkassky et Goldberg [18] pour le problème de débit maximum (code source disponible dans [77]). Nous avons ensuite résolu le programme mixte suivant avec Ilog-CPLEX 12.6 [78].

$$\text{Max} \sum_{v \in V} (x_v + y_v)$$

subject to

$$x_u + y_v \leq 1 \quad , \quad x_v + y_u \leq 1 \qquad \forall (uv) \in E \tag{5.0.9}$$

$$x_v + y_v \leq 1 \qquad \forall v \in V \tag{5.0.10}$$

$$\sum_{v \in V} (x_v + y_v) \leq n - \alpha_{min} \tag{5.0.11}$$

$$1 \leq \sum_{v \in V} x_v \leq \lfloor \frac{n - \alpha_{min}}{2} \rfloor \tag{5.0.12}$$

$$1 \leq \sum_{v \in V} y_v \leq \beta(n) \tag{5.0.13}$$

$$1 \leq \sum_{v \in V} x_{ia} \leq \beta(n) \tag{5.0.14}$$

$$x_v, y_v \in \{0, 1\} \qquad \forall v \in V \tag{5.0.15}$$

L'inégalité $\sum_{v \in V} x_v \leq \lfloor \frac{n - \alpha_{min}}{2} \rfloor$ vient de ce qui suit : Soit $\{A, B, C\}$ un séparateur. On peut supposer sans perte de généralité $|A| \leq |B|$. Depuis $|A| + |B| \leq n - \alpha_{min}$, nous avons $|A| \leq \lfloor \frac{n - \alpha_{min}}{2} \rfloor$.

Science les graphes ont des ordres de hauteur, alors ils prennent plus de temps et plus de nœuds de branche pour résoudre notre problème dans la programmation mixte-entière ; alors, nous devons améliorer certaines inégalités. Par conséquent, nous

remplaçons l'inégalité

$$\sum_{v \in V}(x_v + y_v) \leq n - \alpha_{min}$$

par

$$\sum_{v \in V}(x_v + y_v) \leq n - \alpha*_{min} \tag{5.0.16}$$

Ce $\alpha*_{min}$ est le résultat de nos algorithmes en arrangeant chaque $\alpha_i, i \in V$ dans l'ordre croissant et en choisissant α_i a l'ordre $|A^*| + 1$, $|A^*|$ est un résultat de notre algorithme.

Problème de séparateur de sommet st-connecté

La connectivité sommet et aretes sont deux des concepts les plus fondamentaux de la théorie de la fiabilité réseau. Il y a beaucoup d'articles qui ont étudié un cas particulier, le séparateur de sommets de clique, par exemple ([22, 53, 65]). Les algorithmes de cette nature sont connus sous le nom d'algorithmes FPT (Fixed-Parameter Tractable) avec paramètre en tant que taille de la solution. Marx et al. [60] montre que le problème *st* -CVS est NP-Hard, et qu'il est en FPT. Narayanaswamy et Sadagopan [63] montrent que *st* -CVS est NP-Complète sur les graphes de la corde d'au moins 5 et présente un algorithme de temps polynomial pour *st* -CVS sur les graphes de la 4ème corde. Ils ont également introduit un algorithme pour résoudre le problème et ont prouvé que *st* -CVS paramétré est $W[2]$ - dur. Récemment, ce problème a été étudié par [56] en tant que complexité et point de vue polyédrique. Un graphe simple non dirigé G est une paire (V, E) où V est l'ensemble des sommets et E est l'ensemble des arêtes. Nous notons $n := \mid V \mid$ le nombre de sommets et $m := \mid E \mid$ le nombre d'arêtes. De plus, chaque arête $\{u, v\}$ est notée uv. Pour tout $S, T \subseteq V$ on a $\delta(S) : \delta(S, T) := \{uv \in E : u \in S, v \in T\}$, $\delta(S) := \delta(S, V \setminus S)$ et $E(S) := \{ij \in E : i, j \in S\}$.. Pour tout $F \subseteq E$, nous mettons $V(F) := \{u, v \in V : uv \in F\}$. De plus, nous mettons $N(S) := \{u \in V \setminus S : \exists v \in S, uv \in E\}$. Pour chaque sommet $u \in V$, nous définissons $\delta(u) := \delta(\{u\})$ l'ensemble des arêtes incidentes au sommet u, $N(u) := \{v \in V : uv \in E\}$ le voisinage du sommet u et $d(u) := \mid \delta(u) \mid$. Le degré du sommet u. Pour les graphes simples, nous avons $d(u) = \mid N(u) \mid$. Pour tout S $subseteqV$, on note $G[S]$ le graphe induit par S. Pour tout $S \subseteq V$, pour tout $u \in V$, on pose $NS(U) := N(u) \cap S$ le voisinage du sommet u in $G[S]$ et $dS(U) := \mid NS(U) \mid$ Le degré du sommet u dans $G[S]$.

Soit s et t deux sommets disjoints de V, non adjacents. Un séparateur st -connecté dans le graphe G est un ensemble $S \subseteq V \setminus \{s,t\}$ tel que

(i) il n'y a plus de chemins entre s et t dans $G[G \setminus S]$, et

(ii) le graphe $G[S]$ est connecté.

Nous supposerons que G est connecté. De plus, pour tout séparateur S dans G, il existe au moins une bi-partition (A,B) de $V \setminus S$ telle que $s \in A$, $t \in B$, et $\delta(A,B) = \phi$.

Formulations

Dans cette section, nous allons donner quelques formulations.

Formulation naturelle (NF)

Le $(s,t) - CVS$ équivaut à résoudre le programme linéaire

$$min\{\sum_{v \in V}(w_v x_v),\ x \in P_{st}(G)\}.$$

Le problème de séparateur st -connected peut être formulé comme suit :
$min \sum_{v \in V}(x_v)$
s.t.

$$x_s = x_t = 0 \tag{5.0.17}$$

$$X(V_I(P_{st})) \geq 1 \ \ \forall P_{st} \in \Gamma_{st}, \tag{5.0.18}$$

$$\begin{aligned} X(N(U)) \geq x_u + x_v - 1 \quad & \forall u,v \subseteq V \setminus \{s,t\} with\ uv \notin E, \\ & \forall U \subseteq V \setminus \{s,t\} with\ u \in U, v \notin U \cup N(U) \end{aligned} \tag{5.0.19}$$

$$x_v \in \{0,1\} \ \ \forall v \in V. \tag{5.0.20}$$

où P_{st} est un chemin entre s et t, $V_I(P_{st}) \equiv V(P_{st} \setminus \{s,t\}$ les sommets dans le chemin interne P_{st} et Γ_{st} est la famille de tous les chemins entre s et t, et $N(u)$ est le voisinage du sommet u.

Les inégalités (4.6.18) signifient qu'à partir de n'importe quel chemin entre s et t, il existe au moins un sommet du chemin interne dans S.
Les inégalités (5.0.19) nous donnent la garantie que le séparateur est connecté.

Formulation étendue (EF)

min $x(V)$,
s.t.

$$x_s = x_t = 0 \tag{5.0.21}$$

$$X(V_I(P_{st})) \geq 1 \ \ \forall P_{st} \in \Gamma_{st}, \tag{5.0.22}$$

$$z_e \leq x_u \ \ \forall u \in V, \ \ \forall e \in \delta(u), \tag{5.0.23}$$

$$z_e \geq x_u + x_v - 1 \ \ \forall e := uv \in E \setminus (\delta(s) \cup \delta(t)), \tag{5.0.24}$$

$$z(\delta(U)) \geq x_u + x_v - 1 \ \ \forall u, v \in V \setminus \{s,t\}, uv \notin E, \ and \ \forall U \subset V \setminus \{s,t\}, u \in U, v \notin U, \tag{5.0.25}$$

$$z_e \in \{0,1\} \ \ \forall e \in E, \tag{5.0.26}$$

$$x_v \in \{0,1\} \ \ \forall v \in V. \tag{5.0.27}$$

formulation étendue d'arbre (TEF)

min $x(V)$,

s.t.

$$x_s = x_t = 0 \tag{5.0.28}$$

$$x(V(P_{st})) \geq 1 \ \ \forall P_{st} \in \Gamma_{st}, \tag{5.0.29}$$

$$y_e \leq x_u \ \ \forall u \in V, \forall e \in \delta(u), \tag{5.0.30}$$

$$y(E) = \mid V \mid -1, \tag{5.0.31}$$

$$y(\delta(U)) \geq x_u + x_v - 1 \qquad \forall u, v \in V \setminus \{s,t\}, uv \notin E, \ and \ \forall U \subset V, u \in U, v \notin U, \tag{5.0.32}$$

$$0 \leq x_v \leq 1 \ \ \forall v \in V, \tag{5.0.33}$$

$$0 \leq y_e \leq 1 \ \ \forall e \in E, \tag{5.0.34}$$

$$x_v \in \{0,1\} \ \ \forall v \in V. \tag{5.0.35}$$

Heuristique

Dans cette section, nous introduisons un algorithme pour trouver un ensemble de séparateur connecté en peu de temps. Cet algorithme dépend de deux étapes. la première étape consiste à trouver un ensemble de séparateur S en résolvant le problème (P1) (dans la prochaine sous-section) la programmation entière. La deuxième étape consiste à vérifier la solution S, si elle est connectée, c'est fait. Sinon, nous essayons de trouver des sommets (un ou plusieurs) qui rendent S connecté.
Soit $C_1, ..., C_k$ les composantes connexes du sous-graphe $G(S)$, et soit $u \notin S \cup \{s,t\}$, nous définissons C_i^u pour $i = 1, ..., k$ comme suit : $C_i^u = 1$ si $(uv) \in E$ pour un sommet $v \in C_i$, $C_i^u = 0$ dans le cas contraire.
Nous fournissons des détails pour notre algorithme pour CVSP.

Algorithm 5 algorithme 3

1: *Résoudre P1 pour trouver S.*
2: **if** *S est connecté* **then STOP**
3: *Trouver des composants connexes de S* $(C_1, ..., C_k)$.
4: *Trouver* $u^* \notin S \cup \{s,t\}$ *tel due* $\sum_{i=1}^{k} C_i^{u^*}$ *est maximum,* $S := S \cup \{u^*\}$
5: **goto** *2*

Bibliographie

[1] K. Andreev and R. Racke, *On Balanced Graph Partitioning.*, Theory of Computing Systems,n. 39(6),(2006), pp. 929-939.

[2] M. Armbruster , *Branch-and-Cut for a Semidefinite Relaxation of Large-Scale Minimum Bisection Problems.*, PhD thesis, 2007.

[3] M. Armbruster ,M. Fugenschuh ,C. Helmberg and A. Martin, *A Comparative Study of Linear and Semidefinite Branch-and-Cut Methods for Solving the Minimum Graph Bisection Problem.*, In Proceedings of the 13th International Conference on Integer Programming and Combinatorial Optimization,n. 5035 of LNCS,(2008), pp. 112-124.

[4] A. Arulselvan, C.W. Commander, L. Elefteriadou, and P.M. Pardalos , *Detecting critical nodes in sparse graphs.*, Comput. Oper. Res,n. 36(7), (2009),pp. 2193-2200.

[5] E. Balas and C. de Souza, *The vertex separator problem : a polyhedral investigation*, Mathematical Programming, n. 3, 103(2005), pp. 583–608.

[6] C. Berge, *Graphs and Hypergraphs.*, North-Holland, Amsterdam, (1973).

[7] A. Berger ,A. Grigoriv and R. der Zwaan, *Complexity and approximability of the k-way vertex cut.*, Networks ,n. 63(2),(2014), pp. 170-178.

[8] W. Ben-Ameur and Mohamed Didi Biha , *On the minimum cut separator problem.* , Networks, n. 59(1) ,(2012), pp. 30-36.

[9] W. Ben-Ameur, M. Mohamed-Sidi, and J. Neto, *The k-Separator Problem.*, International Computing and Combinatorics Conference ,(2013), pp. 337-348.

[10] U. Benlic and J.K. Hao, *Breakout local search for the vertex separator problem*, Proceedings of the Twenty-Third international joint conference on Artificial Intelligence, August(2013),Beijing,China.

[11] L. Brunetta ,M. Conforti and G. Rinaldi, *A Branch-and-Cut Algorithm for the Equicut Problem.*, Mathematical Programming,n. 78(2),(1997), pp. 243-263.

[12] E. G. Boman, K. D. Devine, and S. Rajamanickam, *Scalable Matrix Computations on Large Scale-Free Graphs Using 2D Graph Partitioning.*, In ACM/IEEE Conference for High Performance Computing,Networking, Storage and Analysis (SC).,(2013).

[13] T.N. Bui and C. Jones, *Finding Good Approximate Vertex and Edge Partitions is NP-Hard*, Information Processing Letters,42(1992), pp. 153–159.

[14] G. Calinescu, C. G. Fernandes, and B. Reed, *Multicuts in unweighted graphs and digraphs with bounded degree and bounded tree-width.*, ournal of Algorithms,n. 48(2),(2003), pp. 333-359.

[15] K. S. Camilus and V. K. Govindan, *A Review on Graph Based Segmentation.*, IJIGSP,n. 4,(2012), pp. 1-13.

[16] V.F. Cavalcante and C.C. de Souza, *Lagrangian relaxation and cutting planes for the vertex separator problem*, , in Combinatorics, Algorithms, Probabilistic and Experimental Methodologies,Lecture Notes in Computer Science, B. Chen, M. Paterson, and G. Zhang, eds., Springer-Verlag,(2007),pp. 471-482.

[17] G. Chartrand and L. Lesniak, *Graphs and digraphs.*, Wadsworth and Brooks, (1986).

[18] B.V. Cherkassky and A.V. Goldberg, *On Implementing Push-Relabel Method for the Maximum Flow Problem*, Algorithmica, 19(1997), pp. 390-410.

[19] B.V. Cherkassky ; A.V. Goldberg and T. Radzik , *Shortest paths algorithms : theory and experimental evaluation.*, In Proceedings of the third annual ACM sumposium on Theory of computing, (1971), pp. 151–158.

[20] J. Cong and J. Shinnerl., *Multilevel Optimization in VLSICAD.*, Springer,(2003).

[21] S.A. Cook, *The complexity of theorem-proving procedures*, IMathematical Programming.,n. 73(2)(1996), pp. 129–174.

[22] E. Dahlhaus ; M. Karpinski ; and M.B. Novick, *Fast parallel algorithms for the clique separator decomposition*, In Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms :SODA 90, Philadelphia, PA, USA, (1990), pp. 244–251.

[23] D. Delling ,V. Goldberg ,I. Razenshteyn and F. Werneck, *Exact Combinatorial Branch-and-Bound for Graph Bisection.*, In Proceedings of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX 12),(2012), pp. 30-44.

[24] D. Delling and F. Werneck, *Better Bounds for Graph Bisection.* , In Proceedings of the 20th European Symposium on Algorithms, n. 7501 of LNCS,(2012), pp. 407-418.

[25] C.C. de Souza and V.F. Cavalcante, *Exact algorithms for the vertex separator problem in graphs*, Networks an international journal,n. 3,57(2011), pp. 212–230.

[26] C. de Souza and E. Balas, *The vertex separator problem : algorithms and computations*, Mathematical Programming, n. 3, 103(2005), pp. 609–631.

[27] J. Diaz ,M.D. Penrose, J. Petit and M. Serna, *Approximating layout problems on random geometric graphs*, Algorithms, 39(2001), pp. 267-292.

[28] M. Didi-Biha and M.J. Meurs, *An exact algorithm for solving the vertex separator problem*, Journal of Global Optimization,n. 3,49(2011), pp. 425–434.

[29] M. Di Summa, A. Grosso, M. Locatelli, M, *Branch and cut algorithms for detecting critical nodes in undirected graphs.* , Computational Optimization and Applications, n. 53(3),(2012), pp. 649-680.

[30] J. Edmonds, *Convers and packings in a familly of sets.*, Bulletin of the American Mathematical Society,n. 68, 5(1962), pp. 494–499.

[31] A. Feldmann and P. Widmayer, *An $O(n^4)$ Time Algorithm to Compute the Bisection Width of Solid Grid Graphs.* , In Proceedings of the 19th European Conference on Algorithms, n. 6942 of LNCS,(2011), pp. 143-154.

[32] A. Felner , *Finding Optimal Solutions to the Graph Partitioning Problem with Heuristic Search.* , Annals of Mathematics and Artificial Intelligence, n. 45,(2005), pp. 293-322.

[33] C. Ferreira ,A. Martin , C. De Souza ,R. Weismantel and L. Wolsey, *The Node Capacitated Graph Partitioning Problem : A Computational Study.* , Mathematical Programming, n. 81(2),(1998), pp. 229-256.

[34] C. M. Fiduccia and R. M. Mattheyses, *A Linear-Time Heuristic for Improving Network Partitions.* , In Proceedings of the 19th Conference on Design Automation,(1982), pp. 175-182.

[35] L.R. Ford and D.R. Fulkerson , *Maximal flow through a network.*, Canadian Journal of Mathematics.,n. 8,(1956), pp. 399–404.

[36] J. Fukuyama, *NP-completeness of the planar separator problems*, Journal of Graph Algorithms and Applications, 4(2006), pp. 317–328.

[37] M.R. Garey and D.S. Johnson, *Computers and Intractabiliy*, W.H. Freeman and Compagny, (1979).

[38] P.W. Goldberg ,M.C. Golumbic,H. Kaplan and R. Shamir, *Four strikes against physical mapping of DNA*, Journal of Computing Biology, 12(1995), pp. 139–152.

[39] J. Gusted, *On the pathwidth of chordal graphs*, Discrete Applied Mathemastics, 45(1993), pp. 233–248.

[40] W. Hager and J.T. Hungerford, *Continuous quadratic programming formulations of optimization problems on graphs*, European Journal of Operational Research, 240(2014), pp. 328–337.

[41] W. Hager ,D. Phan and H. Zhang, *An Exact Algorithm for Graph Partitioning.* , Mathematical Programming, n. 137,(2013), pp. 531-556.

[42] T.E. Harris and F.S. Ross, *Fundamentals of a Method for Evaluating Rail Net Capacities.*, Research Memorandum. Rand Corporation.,(1955).

[43] C. Helmberg and F. Rendl, *A spectral bundle method for semidefinite programming*, SIAM J. Numer. Anal., 36(1979), pp. 177–189.

[44] L. Hyafil and R. Rivest, *Graph partitioning and constructing optimal decision trees are polynomial complete problems.*, Technical Report Rapport de Recherche, n. 33, (1973), IRIA - Laboratoire de Recherche en Informatique et Automatique.

[45] S. Jesú s , J.P. Juan and D. Abraham, *Combing inteensification and diversification strategies in VSN.An application to the Vertex Separator problem*, Computers and Operations Research,52(2014), pp. 209–219.

[46] B. Junker and F. Schreiber., *Analysis of Biological Networks.* , Wiley, (2008).

[47] S. Karisch ,F. Rendl and J. Clausen, *Solving Graph Bisection Problems with Semidefinite Programming.* , INFORMS Journal on Computing, n. 12(3),(2000), pp. 177-191.

[48] R. M. Karp, *Reducibility Among Combinatorial Problems.*, In R. E. Miller and J. W. Thatcher (editors), Complexity of Computer Computations, New York,(1972), pp. 85–103.

[49] B.W. Kernighan and S. Lin, *An Efficient Heuristic Procedure for Partitioning Graphs* , The Bell System Technical Journal, n. 49(1),(1970), pp. 291-307.

[50] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI Physical Design - From Graph Partitioning to Timing Closure* , Springer, (2011).

[51] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter, *Distributed Time-Dependent Contraction Hierarchies.* , In 9th Symposium on Experimental Algorithms, n. 6049 of LNCS,(2010), pp. 83-93.

[52] U. Lauther, *An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background.* , In Munster GI-Days, (2004).

[53] H.G. Leimer, *Optimal decomposition by clique separators*, Discrete Math., n. 113,(1993), pp. 99-123.

[54] V. V. Lepin, *Solving the weighted k-separator problem in graphs with specific modules* , Trudy Instituta Matematiki,Russia, n. 24(1),(2016), pp. 61-74.

[55] R. J. Lipton and R. E. Tarjan, *A separator theorem for planar graphs* , SIAM Journal on Applied Mathematics, n. 36,(1979), pp. 177-189.

[56] Y. Magnouche , *The multi-terminal vertex separator problem :Complexity, Polyhedra and Algorithms.* , Ph.D thesis.Dauphine Universtie.Paris,(2017).

[57] Y. Magnouche and S. Martin, *The Multi-terminal Vertex Separator Problem : Polytope Characterization and TDI-ness* , International Symposium on Combinatorial Optimization,n. 43(2),(2014).

[58] Y. Magnouche ,A.R. Mahjoub and S. Martin, *The Multi-terminal Vertex Separator Problem : Extended formulations and Branch and Cut and Price* , converence IEEE,(2016).

[59] A.R. Mahjoub, *Concepts of Combinatorial Optimization,Polyhedral Approaches.*, ,Wiley Online Library ,(2013), pp. 261–324.

[60] D. Marx, B. O'sullivan, and I. Razgon, *Finding small separators in linear time via treewidth reduction*, ACM Trans. Algorithms, n. 9(4),(2013), pp. 1-30.

[61] D. Marx and I. Razgon, *Fixed-parameter tractability of multicut parameterized by the size of the cutset* , siam journal on computing,ISCO,(2016),PP. 320-331.

[62] R. Mondaini, *Biomat 2009* , International Symposium on Mathematical and Computational Biology,Brasilia, Brazil, August 2009, World Scientific,(2010).

[63] N.S. Narayanaswamy and N. Sadagopan, *Connected (s, t)-Vertex Separator Parameterized by Chordality*, Journal of Graph Algorithms and Applications, n. 1, 19(2015), pp. 549–565.

[64] G. Nemhauser and L. Wolsey , *Integer and Combinatorial Optimization* ., Wiley-Interscience, New York,(1988).

[65] K.G. Olesen ;and A.L. Madsen, *Maximal prime subgraph decomposition of Bayesian networks*, IEEE Trans. Syst. Man Cybernet, n. 32,(2002), pp. 21-31.

[66] C. Papadopoulos, *Restricted vertex multicut on permutation graphs.*, Discrete Applied Mathematics., n. 160(12),(2012), pp. 1791-1797.

[67] B. Peng, L. Zhang, and D. Zhang, *A survey of Graph Theoretical Approaches to Image Segmentation.*, Pattern Recognition., n. 46(3),(2013), pp. 1020-1038.

[68] J. Sanchez-Oro , N. Mladenovic and A. Duarte, *General variable neighborhood search for computing graph separators*, Optimization Letters,,(2014), pp. 1–21.

[69] A. Schrijver, *Combinatorial Optimization :Polyhedra and Efficiency .*, Algorithms and Combinatorics,n. 24,Springer(2003).

[70] M. Sellmann ,N. Sensen and L. Timajev, *Multicommodity Flow Approximation used for Exact Graph Partitioning.*, In Proceedings of the 11th European Symposium on Algorithms., n. 2832 of LNCS,(2003), pp. 752-764.

[71] N. Sensen , *Lower Bounds and Exact Algorithms for the Graph Partitioning Problem Using Multicommodity Flows.* , In Proceedings of the 9th European Symposium on Algorithms, n. 2161 of LNCS,(2001), pp. 391-403.

[72] F. Spiksma, M. Oosten, and J. Rutten , *Disconnecting graphs by removing vertices :a polyhedral approach.* , Statistica Neerlandica, n. 61 ,(2007), pp. 35-60.

[73] M. Ventresca and D. Aleman , *Efficiently identifying critical nodes in large complex networks.* , Computational Social Networks, n. 2(6) ,(2015).

[74] J. Vygen and B. Korte , *Combinatorial optimization : theory and algorithms.* , Springer., (2005).

[75] Z. Zhang and Z. Shao, *An improved K-opt local search algorithm for the vertex separator problem*, Journal of Computational and Theoretical Nanoscience, n. 11, 12(2015), pp. 4942–4958.

[76] HTTP ://www.ic.unicamp.br/~cid/Problem-instances/VSP.html

[77] HTTP ://www.avglab.com/andrew/soft.html

[78] HTTP ://www.ilog.com

[79] HTTP ://www.optsicom.es/maxcut/♯instances.com

Theoretical and numerical studies on the graph partitioning problem

Abstract :

Given G=(V,E) a connected undirected graph and a positive integer β(n), where n is number of vertices, the vertex separator problem (VSP) is to find a partition of V into three classes A,B and C such that there is no edge between A and B, max{|A|,|B|} less than or equal β(n), and |C| is minimum. In this thesis, we consider an integer programming formulation for this problem. We describe some valid inequalties and using these results to develop algorithms based on neighborhood scheme.
We also study *st*-connected vertex separator problem. Let *s* and *t* be two disjoint vertices of V, not adjacent. A *st*-connected separator in the graph G is a subset S of V\{*s,t*} such that there are no more paths between *s* and *t* in G[G\S] and the graph G[S] is connected . The st-connected vertex sperator problem consists in finding such subset with minimum cardinality. We propose three formulations for this problem and give some valid inequalities for the polyhedron associated with this problem. We develop also an efficient heuristic to solve this problem.

Keywords: Combinatorial optimization, Partitioning graph, Heuristics, Greedy algorithm, Branch and bound algorithm.

Études théoriques et numériques du problème de partitionnement dans un graphe

Résumé:

Étant donné G = (V, E) un graphe non orienté connexe et un entier positif β (n), où n est le nombre de sommets de G, le problème du séparateur (VSP) consiste à trouver une partition de V en trois classes A, B et C de sorte qu'il n'y a pas d'arêtes entre A et B, max {| A |, | B |} est inférieur ou égal à β (n) et | C | est minimum. Dans cette thèse, nous considérons une modélisation du problème sous la forme d'un programme linéaire en nombres entiers. Nous décrivons certaines inégalités valides et et développons des algorithmes basés sur un schéma de voisinage.
Nous étudions également le problème du st-séparateur connexe. Soient s et t deux sommets de V non adjacents. Un st-séparateur connexe dans le graphe G est un sous-ensemble S de V \ {s, t} qui induit un sous-graphe connexe et dont la suppression déconnecte s de t. Il s'agit de déterminer un st-séparateur de cardinalité minimum. Nous proposons trois formulations pour ce problème et donnons des inégalités valides du polyèdre associé à ce problème. Nous présentons aussi une heuristique efficace pour résoudre ce problème.

Mots-clés: optimisation combinatoire, partitionnement, Heuristique, algorithme glouton, Séparation et évaluation.