POLITECNICO DI MILANO
Facoltà di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica

BICLIQUE COMPLETION PROBLEM: MODELS AND ALGORITHMS

Relatore: Prof. Francesco MAFFIOLI
Correlatore: Ing. Stefano GUALANDI

Tesi di Laurea di:
Claudio Patrizio MAGNI
Matr. n. 720827

Anno Accademico 2008–2009

Sommario

Il presente lavoro di tesi si inserisce nell'ambito di ricerca dell'Ottimizzazione Discreta e riguarda, in particolare, modelli e metodi per problemi di ottimizzazione combinatoria. L'argomentazione è incentrata su un problema di completamento di biclique su grafo bipartito, denominato *k-Clustering minimum Biclique Completion*. Il campo applicativo principale a cui si è fatto riferimento è quello delle telecomunicazioni e riguarda l'aggregazione di sessioni per trasmissioni multicast. L'obiettivo principale del lavoro consiste nel progettare ed implementare metodi di soluzione efficaci, sia euristici che esatti.

Il primo passo corrisponde alla formalizzazione del problema in modelli matematici di programmazione intera. Dei vari modelli si sono studiate le proprietà per mettere in luce le difficoltà che sarebbero potute nascere nell'ottenere una soluzione ottima. I modelli sono stati sottoposti a test mediante un risolutore commerciale. Successivamente si è progettato una euristica di ricerca locale, basata su intorni variabili e ricerca nella regione inammissibile, per trovare soluzioni buone in un tempo limitato. A questo primo metodo è seguito un algoritmo per ottenere un buon rilassamento del problema. Si è preferito un metodo di generazione di colonne, con particolare attenzione al problema di Pricing, per il quale un'euristica è stata sviluppata. I due metodi appena descritti hanno permesso di realizzare un algoritmo di Branch-and-Price per la soluzione esatta del problema. Numerosi test sono stati eseguiti per tutti gli algoritmi implementati, su diversi tipi di istanze e con diverse varianti.

I risultati ottenuti hanno dimostrato l'efficacia dei metodi sviluppati, soprattutto per quanto concerne l'euristica di ricerca locale. Il metodo di Branch-and-Price ha permesso di risolvere all'ottimo istanze di dimensioni maggiori rispetto a migliori risultati trovati in letteratura. Diversi aspetti del lavoro hanno rivelato spunti per ricerche future.

Ringraziamenti

Desidero ringraziare il prof. Maffioli per avermi introdotto ad un problema matematico che ha suscitato notevolmente il mio interesse e per avermi saputo consigliare in modo proficuo durante il percorso di tesi. Il ringraziamento più grande va sicuramente a Stefano Gualandi: il suo aiuto è stato fondamentale nel portare a compimento quest'opera. Si è sempre dimostrato disponibile ed entusiasta e mi ha fatto scoprire aspetti affascinanti in ciò che ho affrontato.

Ringrazio i compagni di corso a me più cari: Mae, Maggio e Padu. Grazie a loro i miei anni universitari sono trascorsi in modo piacevole e gli scogli che mi si sono presentati sono stati più facili da superare.

Ringrazio Francesca per avermi supportato nei momenti di maggior difficoltà, durante i giorni precedenti alla conclusione di questo lavoro.

Infine ringrazio i miei genitori per avermi permesso di raggiungere questo traguardo e per la serenità con cui mi hanno accompagnato.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

Here follows a list of definitions of all the acronyms that appear in the document.

k-CmBCP *k-Clustering minimum Biclique Completion Problem*

MPP *Multicast Partition Problem*

LP *Linear Programming*

IP *Integer Programming*

MIP *Mixed Integer Programming*

LS *Local Search*

TS *Tabu Search*

VNS *Variable Neighborhood Search*

VSS *Variable Space Search*

VNUS *Variable Neighborhood Unfeasible Search*

CP *Constraint Programming*

GRASP *Greedy Randomly Adaptive Search Procedure*

B&B *Branch-and-Bound*

B&C *Branch-and-Cut*

B&P *Branch-and-Price*

LB *Lower Bound*

UB *Upper Bound*

MP *Master Problem*

RMP *Restricted Master Problem*

PP *Pricing Problem*

Chapter 1

Introduction

The work presented in this thesis belongs to the research field of Discrete Optimization. In particular we focus on the framework that deals with combinatorial optimization problems and specific methods to solve them. The application ground to which we refer the most is Telecommunications, in which optimization represents a key step when dealing with systems of ever-increasing complexity and dimension.

The objective of the thesis is to study a recently emerged problem, consisting of node aggregation on graphs, and to develop an efficient solution method for it. In designing, analyzing and testing the system we keep an abstract vision of the problem, but we also consider some possible implications related to concrete cases in real life. Generally, two joint efforts from Operations Research and Algorithm Design are combined to solve an optimization problem. Even though a huge amount of problems have been defined and studied in the past years, new ones continue to arise from practical situations in many professional environments. Furthermore, problems often happens to be NP-hard (see [1]): a traditional approach offers poor results, since no algorithm with a number of steps polynomial in the size of the instances is known for solving them.

The optimization problem we are going to deal with arose in 2006 in the telecommunication field. It consists in the aggregation of several multicast sessions. This application is likely to be the source from which many studies will be undertaken in the near future, because the new trend in technology continually increases the need of sending stream of digital information to a great number of users. The very fast evolution of telecommunication network allows wide flexibility in the deployment of various new services. However it becomes more and more difficult to organize all these new services

and their underlying protocols in an efficient and optimized way, especially when the services interconnect heterogeneous entities in a multicast fashion. Some optimization problems deal with the management and allocation of the multicast trees (i.e. tree interconnecting all the nodes of a given multicast session). However few papers are concerned with the problem of aggregating several multicast sessions in a restricted number of trees. This problem is considered in [2] and was already introduced in [3, 4].

Conforming to a traditional engineering approach, a formal definition of the problem is derived from the application field. All the subsequent work is then applied to the abstract definition. This helps to visualize different interpretations and applications of what will be developed.

At first the problem is formulated in terms of mathematical programming. Several models are outlined, each one with its own peculiarities, as well as pros and cons. Then a heuristic framework is developed, in the attempt to deliver a good solution even for very big instances. Many concepts and methods found in the literature are investigated and some chosen for the final algorithm. Constructive heuristics are used to obtain a starting solution to which we apply a local search method to find a better integer solution. This solution, however, is not guaranteed to be optimal. A relaxation method is developed in order to have an approximate estimate of the possible range of values of the optimal solution. Then the heuristic and the relaxation are combined in a branching method like Branch-and-Bound to find the optimal solution.

The implementation is a delicate matter. First a decision has to be made about which language or system to use, taking into account many variables, like efficiency, ease of use and reliability. Second, the formal definition of a method often misses many details that must be faced when writing the code of the program.

Subsequently, extensive testing of the models and the algorithms is performed, in order to understand which work, which do not and to give a measure to their effectiveness. Solution methods often provide some parameters that can be changed to tune their behavior. Other times, different choices, both methodological and practical, are tested to show which is the winning one. The data we have used for testing come from different sources and present different structures, so that the performance of the programs can be exposed under many aspects.

This work allow to gain insight into the problem and the implemented methods; several conclusions can be derived from it. Also the work reveals

some interesting areas that can lead future studies based on this subject.

1.1 Goals

The main goal of the work is to find a good algorithm to solve the k-CmBCP, that will be presented in the next chapter. The idea is to restrict the range of values of the optimal solution as much as possible by means of an upper bound and a lower bound. Following a traditional approach, the upper bound is given by a heuristic, while the lower bound is given by a relaxation method. Based on the dimension of the instance and the availability of resources, optimality can be achieved, or pursued, with a Branch-and-Bound method. The solution method and the relative program should possess several qualities that will be outlined further in the document.

A secondary goal is to discover useful properties of the problem, that could make its solution method more efficient, and to derive interesting applications and interpretations, starting from the formal definition of the problem.

Our last intention is to experiment with different heuristic approaches, in order to develop generic and efficient solution methods.

1.2 Structure of the document

In Chapter 2 the problem is presented formally, along with some important applications and some background about graph theory. Chapter 3 lists all the mathematical programming formulations of the problem: several models and their main properties are outlined. Chapter 4 contains the core of the work, where the solution methods are described: an introduction to the underlying theory is provided, than each subsequent section deals with a specific part of the whole framework. Chapter 5 describes the implementation of the methods. Before that, we introduce the reader to the Comet language, a powerful optimization system that has been chosen in order to implement the majority of the algorithms: the idea is to provide a short tutorial of the language and to show the most remarkable features. Chapter 6 is dedicated to results obtained by testing the various programs. Finally, in the Chapter 7 we report the conclusions that can be grasped from this work, together with suggestions about future work on the subject.

Chapter 2

The k-Clustering Minimum Biclique Completion Problem

This chapter will present the problem and some real-world applications. First a refresh of the basic graph theory is provided: this will allow to establish the notation used throughout the whole document. For further details about graphs see [5].

2.1 Graph Theory

An *undirected graph* G (generally called simply *graph*) is a pair $\langle V, E\rangle$, where V is a set of vertices, or nodes, and $E \subseteq V \times V$ is a set of edges. An edge is a pair of vertices that are called the *endpoints* of the edge. Two vertices are *adjacent* or *connected* if there is an edge between them. The *neighborhood* $\beta(v, G)$ of a vertex v in $G = \langle V, E\rangle$ is the set of vertices adjacent to v, denoted $\beta(v, G) = \{u \mid \{u, v\} \in E\}$. The neighborhood $\beta(X, G)$ of a set of vertices X in $G = \langle V, E\rangle$ is the set of vertices adjacent to every vertex in X; that is, $\beta(X, G) = \bigcap_{v \in X} \beta(v, G) = \{u \mid u \in V, \text{ and } X \subseteq \beta(u, G)\}$.

If the two vertices defining each edge are a directed pair, the graph is called a *directed graph*, or *digraph*. The edges are preferably called arcs in this case and they are represented as arrows. Considering a directed arc $a = (i, j) \in A$, vertex i is called tail of arc a, whereas vertex j is called head of arc a. The *forward star* of a node i is the set of arcs that have i as tail ($FS(i) = \{(i, j) \in A\}$), while the *backward star* of a node i is the set of

arcs that have i as head ($BS(j) = \{(i,j) \in A\}$). Finally, the *star* of a node i is the union of forward and backward star, that is the set of all the arcs that have at least one endpoint in i ($S(i) = FS(i) \cup BS(i)$). For undirected graphs only the concept of star is meaningful. The *degree* of a node is the cardinality of its star. From now on we shall refer to undirected graphs, since this work is based on them.

A graph can be equivalently described by its adjacency matrix. Let $G = \langle V, E \rangle$ be a graph with $V = \{v_1, v_2, ..., v_p\}$. The *adjacency matrix* $\mathbf{A}$ of G is the $p \times p$ matrix defined by

$$\mathbf{A}[i,j] = \begin{cases} 1 & \text{if } \{i,j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

A graph $G' = \langle V', E' \rangle$ is a *subgraph* of a graph $G = \langle V, E \rangle$ induced by V', if $V' \subseteq V$ and $E' = \{\{i,j\} \in E : i,j \in V'\}$. If $V' \subset V$ we say G' is a proper subgraph of G. If G' is a subgraph of G, we say G is a super-set graph of G', or G contains G'. A graph $G' = \langle V, E' \rangle$, defined by $E' \subseteq E$, is a *partial graph* of G. The two definitions can be combined to obtain a *partial subgraph*.

A graph is *simple* if it has no multiple edges (edges that connect the same two endpoints) and no self-loops (edges with one endpoint identical to the other one). A *complete graph* is a simple graph in which every pair of distinct vertices is connected by an edge. A *clique* in a graph $G = \langle V, E \rangle$ is a subset of the vertex set $C \subseteq V$, such that for every two vertices in C, there exists an edge connecting them. This is equivalent to saying that the subgraph induced by C is complete. The *size of a clique* is the number of vertices it contains.

A graph $G = \langle V, E \rangle$ is a *bipartite graph* if its vertex set V can be partitioned into two disjoint non-empty sets V_1 and V_2, and every edge in E connects a vertex in V_1 and a vertex in V_2. So, there is no edge in E connecting two vertices within V_1 or two vertices within V_2. A bipartite graph G is often denoted as $G = \langle V_1, V_2, E \rangle$. V_1 and V_2 are called, respectively, left and right shore or part.

A bipartite graph $G = \langle V_1, V_2, E \rangle$ is called a *biclique* if, for every $v_1 \in V_1$ and $v_2 \in V_2$, there is an edge between v_1 and v_2. Thus the edge set E of a biclique $G = \langle V_1, V_2, E \rangle$ is completely determined by the two vertex sets V_1 and V_2. We will often omit the edge set and denote a biclique G simply as $G = \langle V_1, V_2 \rangle$.

2.2 The problem

The problem we are going to deal with is the k-Clustering minimum Biclique Completion Problem (k-CmBCP), defined on bipartite graphs. The objective is to group the nodes of the left shore in k clusters, so that the number of edges that must be added to form k bicliques, called *total cost*, is minimum. The clusters must induce a partition of the nodes of the left shore. The graph is undirected.

In other words, whenever a cluster is identified, let us consider all the possible edges connecting the left nodes with the right ones. A biclique requires that all these edges are present in the graph. The missing ones are considered as cost and summed. The objective is to find the way to partition the nodes in order to have the lowest total cost.

Let us now present the formal notation used throughout the document.
The set of left shore nodes is represented by $I = \{1, ..., n\}$.
The set of right shore nodes is represented by $J = \{1, ..., m\}$.
The number of clusters in which the left shore nodes must be grouped is p, where $p < n$. The set of all possible clusters is represented by T. Note that p is exactly the k present in the name of the problem (k-CmBCP): this notation is used to avoid confusion with other k that are going to be introduced in the solution methods. Whenever the context is non-ambiguous k will be used instead.
The set of edges can be denoted by a binary matrix $A = [a_{ij}]$, having one row per left node and one column per right node. $a_{ij} = 1$ indicates an edge between i and j. The matrix represents a subset of $I \times J$.

Other than the matrix, the graph notation can be used: $G = \langle I, J, E \rangle$, where E is the set of edges: $E \subset \{i \in I, j \in J\}$. $\bar{E}$ represents the complement of the set of edges: $\bar{E} = \{\{i, j\} \mid i \in I, j \in J, \{i, j\} \notin E\}$.

Each cluster T_k is defined as a triple $\langle I_k, J_k, E_k \rangle$, with $I_k \subset I$ and $J_k \subset J$: it is the subgraph of G induced by (I_k, J_k). The set $E_k = (I_k \times J_k) \cap E$ is the subset of edges between I_k and J_k. The number of edges of each biclique identified by a cluster k is $|I_k| \times |J_k|$. However each cluster has $|E_k|$ edges. Thus $|I_k| \times |J_k| - |E_k|$ is the cost of cluster k.

We can also define a set E^+ of additional edges ($E^+ \subseteq \{i \in I, j \in J\} \backslash E$) of minimum cardinality, such that each cluster T_k can be completed by edges of E^+ in order to form a complete bipartite subgraph or biclique. In other words, each subgraph in $\langle G, E \cup E^+ \rangle$ induced by (I_k, J_k) is a biclique. Then the problem consists in building p clusters such that each edge belongs

to exactly one cluster and the number of additional edges E^+ needed to transform the p clusters into bicliques is minimum. Furthermore $I_1, ..., I_p$ must induce a partition of I.

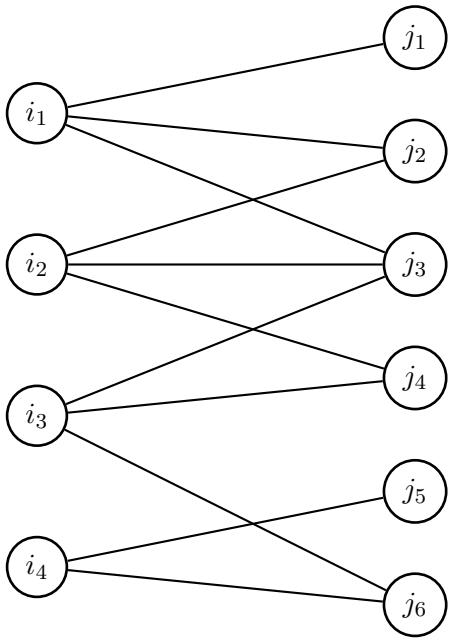


Figure 2.1: Graph representation of the k-CmBCP.

Figure 2.1 shows a bipartite graph representation of the k-CmBCP. In Figure 2.2 we provide two possible solutions of the same instance of Figure 2.1 with two bicliques. The clusters are represented as boxes around the nodes, while the additional edges $\in E^+$ are dotted or dashed. We have also grouped the right shore nodes according to the subgraphs induced by the clusters.

2.3 Notes about the problem

In the previous definition, each additional edge of E^+ counts as one in the total cost. The problem can be adapted to the weighted case, where adding an edge may be more or less costly, depending on some weighting defined for the instance considered. The weight can be defined mainly in two ways: a different weight for each pair (i, j) or a weight for each left shore node i (useful in multicast sessions in telecommunications). Adapting to the first case is far from trivial. Many applications of this variation can be found in the real world. However, the variation with weights on nodes is considered

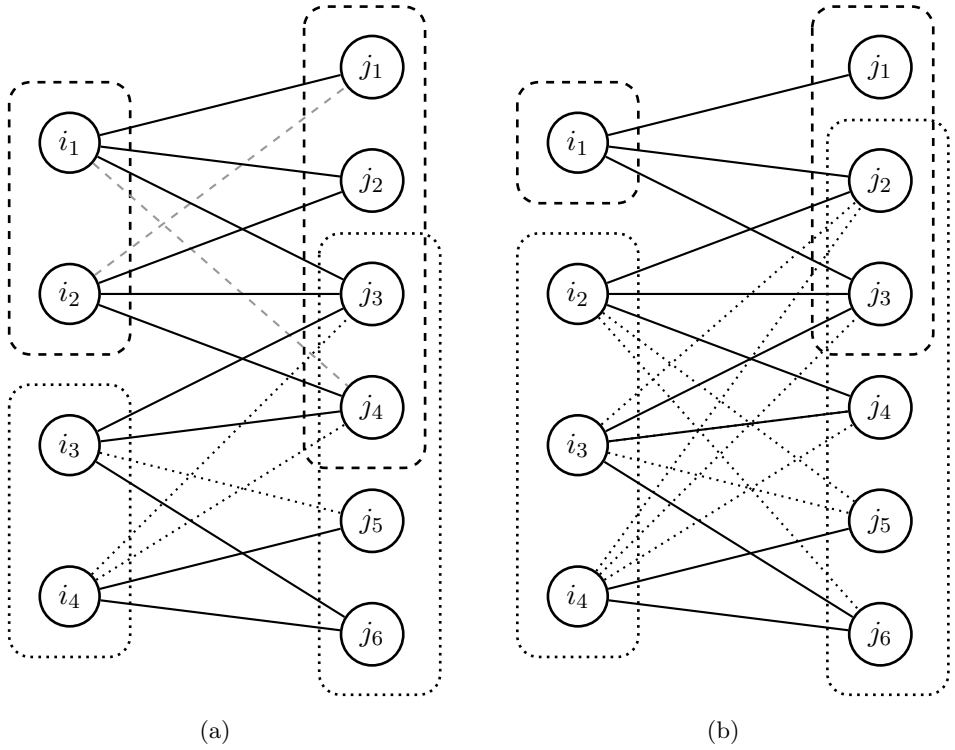


Figure 2.2: Two solutions of the previous instance, with two bicliques.

in Section 4.4, when the B&P is described.

An alternative formulation for the k-CmBCP is obtained by interpreting the problem as a constrained Max-k-Cut problem. It is presented via a Hybrid CP and SDP approach in [6], but it is not going to be considered in the present work.

An easier version of the problem is defined if we omit the partition constraint. It is referred to as *minimum Biclique Cover Problem*, since the only constraint is to cover all the left nodes. It can be seen as a generalization of the k-CmBCP.

k-CmBCP is NP-hard. For the proof see [7]; related works can be found in [8, 9, 10]. Note that the order of the cluster is irrelevant for the objective function: thus a solution is fully characterized by how the nodes are grouped into the sets called clusters. Any model of the problem that gives a explicit order to the bicliques will inevitably present symmetries: in that case the solution space has cardinality p^n.

A final note about the instances is needed: it is required that no nodes are disconnected (i.e. a node with no edges), both from the left and the right

shore of the graph. This requirement has the goal to simplify some issues about modeling the problem. A disconnected node in the left shore, in an optimal solution, will be always inserted in the cluster with the smallest J_k, since it requires all the edges to complete a biclique. A disconnected node in the right shore will be always ignored, because it will not appear in any J_k. Furthermore, if one considers the applications of the problem, disconnected nodes make little sense in reality: this is explained better in section 2.4.

2.3.1 A polynomial special case of the *k*-CmBCP

We consider the special case where each right shore node is connected to a single left shore node. In [7] a polynomial dynamic programming algorithm is presented, along with proof of the properties of this particular case. The worst case complexity of this algorithm is in $O(pn^3)$ time, while the space complexity is $O(pn^2)$. The algorithm proved to be extremely fast even with large instances.

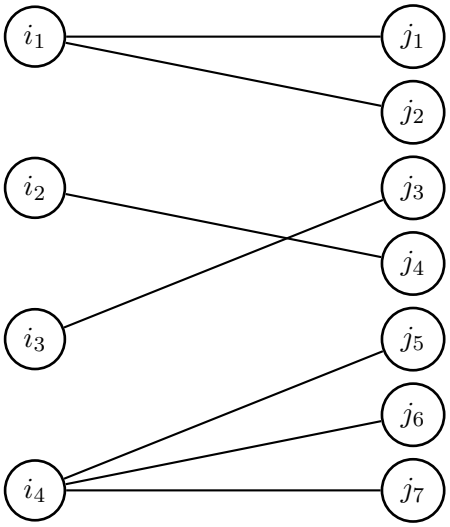


Figure 2.3: An example of the special polynomial case.

2.4 Applications

The application that brought this problem to the attention of researchers deals with telecommunications and was first studied in [7]. The problem consists in the aggregation of multicast sessions. A multicast session is defined as a subset of clients requiring the same information. Besides, each

client can require several multicast sessions. A telecommunication network cannot manage many multicast sessions at the same time. It is hence necessary to group the sessions into a limited number of clusters. The problem then consists in aggregating the sessions into clusters to limit the number of unnecessary information sent to clients. The variant of the problem we shall consider requires the clusters to define a partition on the set of sessions. We shall refer to the problem as *Multicast Partition Problem* (MPP). Since this application was the first one that brought the k-CmBCP to attention, in the document we shall often refer to it instead of the theoretical graph definition. The next paragraphs will make it easier to relate this application to the formal definition of section 2.2.

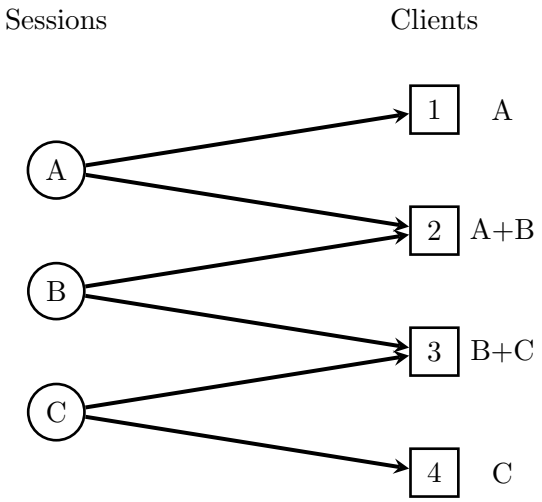


Figure 2.4: An instance of the MPP problem.

According to the previous notations, the set of sessions is represented by $I = \{1, ..., n\}$, while the set of clients is represented by $J = \{1, ..., m\}$. $A = [a_{ij}]$ has one row per session and one column per client. $a_{ij} = 1$ expresses the fact that a session i is requested by a client j. The edges are called *demands* and A is called *demand matrix*.

In a cluster k, all demands are treated as a single multicast group, that is the information of each session in the cluster (set I_k) will be sent to every client that requires at least a demand from at least a session in I_k.

The information sent without a demand requirement is considered as wasted, so the objective is to find the partitioning of the sessions that minimizes the waste. If we call I_k the set of sessions in the cluster k, J_k the set of clients in the cluster k and A_k the set of demands between I_k and J_k, the

number of wasted information units is $|I_k||J_k| - |A_k|$.

Figure 2.4 shows an instance of the MPP. We use the same representation of the k-CmBCP. On the left there are three sessions A, B and C, that send multicast information to the clients (nodes 1, 2, 3 and 4). Each client requires some sessions, which are written at the right of the node. Figure 2.5 shows the same instance when we group sessions A and B together. They now become a single multicast source, that forwards all the information sent by each of its sessions: in this case the union of A and B is shown as thicker edges. Client 1 receives session B even though he did not require it. The same can be said for client 3 with session A.

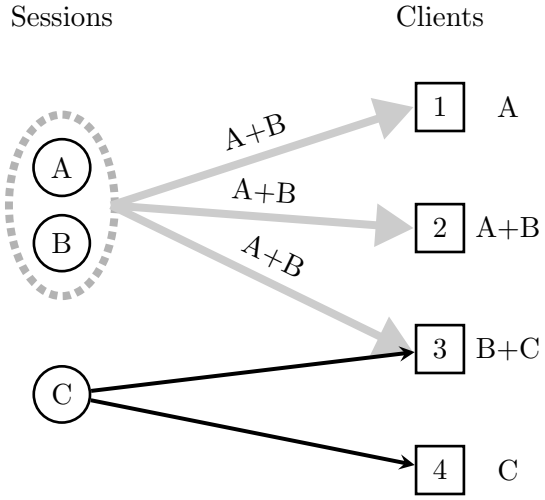


Figure 2.5: Two sessions are grouped in the previous instance.

Other than MPP, a natural application is found in *data mining*, where clustering is a very important method. While there are many variations and interpretations of the problem, in general we can state that an edge is a way to express a connection between left and right shore nodes. If we consider the left nodes as users and the right ones as items, a solution to the problem will try to group together users that appear to have similar connections to the items. Some applications can be: recommender systems, purchase forecasting, computational biology, network security analysis, etc.

Let us consider, for example, the clustering of data by common attribute values. The problem can be expressed as a binary matrix, where the objects are represented by rows and the attributes by columns. A 1 (0) in row i and column j indicates that object i has (does not have) attribute j. Since edges of a bipartite graph are expressed as a binary matrix, the problem is

equivalent to partitioning the graph into bicliques. The k-CmBCP solves this problem when the number of data types is fixed and equal to k and some errors are present in the data. A variation that does not take into account the possibility of erroneous data is the *Biclique Partition Problem*, presented in [11] for the same application.

Chapter 3

Mathematical programming formulations

The problem is an integer programming problem. Several classes of formulations exist:

1. *assignment* formulations, where clusters are indicated explicitly and nodes are assigned to them;
2. *representative* formulations, where clusters are indicated by a representative node and all other nodes are assigned to their "head-boy";
3. *column generation* formulations, where configurations of clusters are considered and the number of variables is exponential.

The first four models we are going to present are combinatorial, with binary variables, while the last one is integer. More on mathematical programming can be found in [12]. For each model the meaning of the constraints is explained. At the end of the chapter some important properties of the models are listed, with their proof.

The notation is taken from the telecommunication problem MPP, but the underlying graph structure is present anyway.

3.1 First model

The first formulation we present is probably the most immediate to derive. It is quadratic, with both linear and non linear constraints. The variables are:

$$x_i^k = \begin{cases} 1 \text{ if session } i \text{ is covered by cluster } k \\ 0 \text{ otherwise} \end{cases}$$

$$y_j^k = \begin{cases} 1 \text{ if client } j \text{ is covered by cluster } k \\ 0 \text{ otherwise} \end{cases}$$

An edge $\{i, j\}$ is paid if it is not present in the edge set E and the nodes that it connects (a session i and a client j) are both in the same cluster. That happens when $x_i^k = y_j^k = 1$ for some $k \in K$. The objective function is obtained by the sum of all the edges that are paid. This is expressed by the product between the x and the y variables, forming a quadratic objective function:

$$min \sum_{\{i,j\} \in \bar{E}} \sum_{k \in K} x_i^k y_j^k$$

A first constraint is needed to cover all the demands. For every edge $\{i, j\}$, we must be sure that there is one and only one cluster that contains both the session i and the client j. A second constraint is required for the partition of the sessions: every session i must be contained in one and only one cluster. Finally a last constraint is needed to avoid the presence of empty cluster, thus using all of the p available clusters.

Quadratic formulation, with a non linear constraint (model-1a)

$$\min \quad \sum_{\{i,j\} \in \bar{E}} \sum_{k \in K} x_i^k y_j^k \tag{3.1}$$

$$\text{s.t.} \quad \sum_{k \in K} x_i^k y_j^k = 1 \qquad \forall \{i, j\} \in E \tag{3.2}$$

$$\sum_{k \in K} x_i^k = 1 \qquad \forall i \in I \tag{3.3}$$

$$\sum_{i \in I} x_i^k \geq 1 \qquad \forall k \in K \tag{3.4}$$

$$x_i^k, y_j^k \in \{0, 1\} \qquad \forall i \in I, j \in J, k \in K \tag{3.5}$$

Constraint 3.2 expresses the covering of all the demands, constraint 3.3 expresses the partition of sessions, while constraint 3.4 ensures that there are no empty clusters.

The formulation can be refined. Constraint 3.4 can be ignored, thanks to the monotonicity of the objective function in the number of clusters (see Section 3.6).

Constraint 3.3 is dominated by 3.2 from below. Thus the equality can be substituted by a less than inequality. Furthermore constraint 3.3 can be removed (as long as 3.2 is kept in the formulation), because the minimization objective will avoid duplicating a session in more than one cluster. This in fact can only make the cost greater or equal than keeping a session in one of the two clusters only (see Section 3.6).

Let us rewrite the first model.

Quadratic formulation, with a non linear constraint (model-1a)

$$\min \quad \sum_{\{i,j\}\in\bar{E}} \sum_{k\in K} x_i^k y_j^k \tag{3.6}$$

$$\text{s.t.} \quad \sum_{k\in K} x_i^k y_j^k = 1 \qquad \forall\, \{i,j\} \in E \tag{3.7}$$

$$x_i^k, y_j^k \in \{0,1\} \qquad \forall\, i \in I, j \in J, k \in K \tag{3.8}$$

Still (model-1a) has non linear constraints. A model of this type is very difficult to solve (see [13, 14] for details). If we keep Constraint 3.3, Constraint 3.7 can be substituted by a linear one:

$$y_j^k \geq x_i^k \qquad \forall\, \{i,j\} \in E, k \in K$$

This constraint in fact forces the y_j^k variable to be 1 whenever there is at least one session in the cluster k that is connected to j. We can write the model with linear constraints.

Quadratic formulation, with linear constraints (model-1b)

$$\min \quad \sum_{\{i,j\}\in\bar{E}} \sum_{k\in K} x_i^k y_j^k \tag{3.9}$$

$$\text{s.t.} \quad y_j^k \geq x_i^k \qquad \forall\, \{i,j\} \in E, k \in K \tag{3.10}$$

$$\sum_{k\in K} x_i^k = 1 \qquad \forall\, i \in I \tag{3.11}$$

$$x_i^k, y_j^k \in \{0,1\} \qquad \forall\, i \in I, j \in J, k \in K \tag{3.12}$$

However there is still another way of expressing Constraint 3.10, that generally requires less inequalities and thus it is preferred:

$$n\, y_j^k \geq \sum_{\{i,j\}\in E} x_i^k \qquad \forall\, j \in J, k \in K$$

where n is the cardinality of set I. Let us write this variant.

Quadratic formulation, with linear constraints (model-1c)

$$\begin{aligned}
\min \quad & \sum_{\{i,j\}\in \bar{E}} \sum_{k\in K} x_i^k y_j^k && && (3.13)\\
\text{s.t.} \quad & n\, y_j^k \geq \sum_{\{i,j\}\in E} x_i^k && \forall\, j \in J, k \in K && (3.14)\\
& \sum_{k\in K} x_i^k = 1 && \forall\, i \in I && (3.15)\\
& x_i^k, y_j^k \in \{0,1\} && \forall\, i \in I, j \in J, k \in K && (3.16)
\end{aligned}$$

From (model-1b) a linear formulation can be obtained through standard linearization techniques. It becomes necessary to introduce a third variable, that expresses the product of x and y:

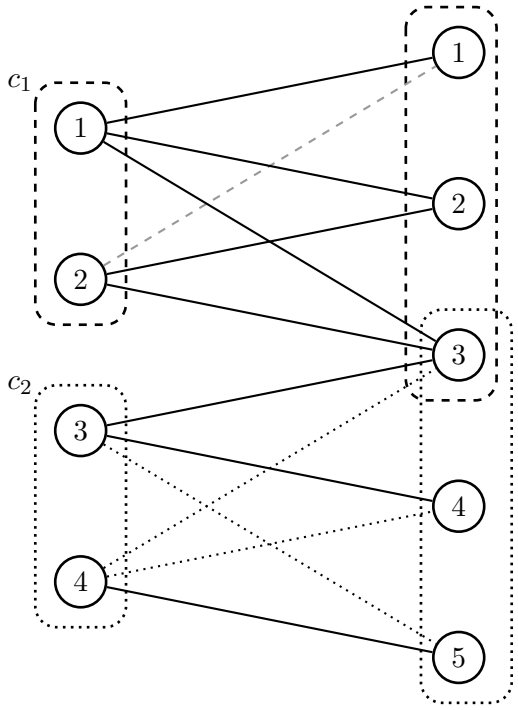
$$z_{ij}^k = \begin{cases} 1 \text{ if session } i \text{ and client } j \text{ are both in cluster } k \\ 0 \text{ otherwise} \end{cases}$$

The objective function is the same as before, with the only difference of variable z instead of the product. Only one set of constraints must be added, with respect to (model-1b): the z variable must effectively express the product between $x \times y$.

Linear formulation (model-1d)

$$\begin{aligned}
\min \quad & \sum_{\{i,j\}\in \bar{E}} \sum_{k\in K} z_{ij}^k && && (3.17)\\
\text{s.t.} \quad & y_j^k \geq x_i^k && \forall\, \{i,j\} \in E, k \in K && (3.18)\\
& \sum_{k\in K} x_i^k = 1 && \forall\, i \in I && (3.19)\\
& z_{ij}^k \geq x_i^k + y_j^k - 1 && \forall\, i \in I, j \in J, k \in K && (3.20)\\
& x_i^k, y_j^k, z_{ij}^k \in \{0,1\} && \forall\, i \in I, j \in J, k \in K && (3.21)
\end{aligned}$$

Similar to the previous formulation, constraint 3.18 expresses the covering of all the demands, while constraint 3.19 expresses the partition of sessions. The last constraint (3.20) is used to represent the product of x and y by means of a linear expression.



$$n = 4,\ m = 5,\ p = 2$$

$$x^1 = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$$

$$x^2 = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$$

$$y^1 = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$$

$$y^2 = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$$

$$z^1 = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$z^2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Figure 3.1: Solution of the k-CmBCP expressed in terms of the first model.

Figure 3.1 shows a solution of the k-CmBCP and lists the values of all the variables.

3.2 Second model

We propose a different non-linear model by interpreting the problem as a *min-max* model. The x_i variables are the same as the previous model, but we introduce a variable z_{ij} with the following meaning:

$$z_{ij} = \begin{cases} 1 \text{ if we pay the edge } \{i,j\} \\ 0 \text{ otherwise} \end{cases}$$

Note that an edge $\{i,j\} \in \bar{E}$ is paid if and only if there is at least a node l, connected to j, in the same cluster of i. If we indicate with $N(j)$

the neighborhood of j ($N(j) = \{i|\{i,j\} \in E\}$), we can write:

$$z_{ij} = \max_{l \in N(j)} \{x_i^k \, x_l^k\}$$

This constraint can be rewritten as a set of $z_{ij} \geq x_i^k \, x_l^k$ constraints, which are easily made linear with: $z_{ij} \geq x_i^k + x_l^k - 1$. The final linear formulation is:

Min-Max model: linear formulation (model-2)

$$\min \quad \sum_{\{i,j\} \in \bar{E}} z_{ij} \tag{3.22}$$

$$\text{s.t.} \quad \sum_{k \in K} x_i^k = 1 \qquad \forall \, i \in I \tag{3.23}$$

$$z_{i,j} \geq x_i^k + x_l^k - 1 \qquad \forall \, \{i,j\} \in \bar{E}, \{l,j\} \in E, k \in K \tag{3.24}$$

$$x_i^k, z_{ij} \in \{0,1\} \qquad \forall \, i \in I, j \in J, k \in K \tag{3.25}$$

Constraint 3.23 expresses the partition of sessions, while Constraint 3.24 is the linearization of the min-max constraint defined above.

3.3 Third model

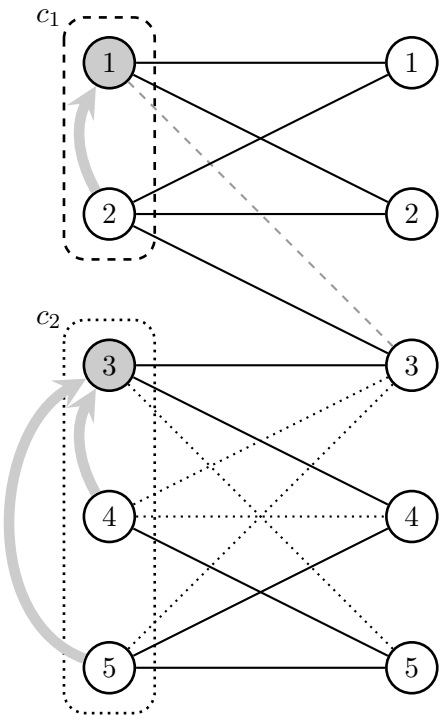
This other model uses the concept of "representative" as the node identifying a cluster. In the previous model clusters were identified by sessions and clients. Here a session node is elected as representative of a cluster. All other session nodes that belongs to the same cluster must express their assignment to the representative. An ordering of the nodes is needed: the first node of a cluster, according to the order, is the representative of the cluster. This model is useful to eliminate symmetries. Figure 3.2 shows how sessions are grouped in this model and how clusters are identified.

The variables are:

$$x_{il} = \begin{cases} 1 \text{ if session } i \text{ is representative of session } l \\ 0 \text{ otherwise} \end{cases}$$

$$z_{ij} = \begin{cases} 1 \text{ if we pay the edge } \{i,j\} \\ 0 \text{ otherwise} \end{cases}$$

The objective function is straightforward, thanks to the z variable:



$n = 5,\ m = 5,\ p = 2$

Node 1 is representative of cluster 1

Node 3 is representative of cluster 2

$$x = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$z = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Figure 3.2: Representation of the "representative" model.

$$min \sum_{\{i,j\} \in \bar{E}} z_{ij}$$

We need a constraint to ensure that p clusters are identified. Another one to assign each session to a single representative, thus to a single cluster. Another constraint is required to correctly link the x variables among themselves: if a session l is assigned to a representative i, than i must be a representative and cannot be a simple node. The last constraint that we need takes care about the z_{ij} variable: it must be set to 1 if the edge $\{i, j\}$ is to be added to complete a cluster (i.e. to obtain a biclique).

Representative model: linear formulation (model-3)

$$\min \quad \sum_{\{i,j\}\in\bar{E}} z_{ij} \tag{3.26}$$

$$\text{s.t.} \quad \sum_{i\in I} x_{ii} = p \tag{3.27}$$

$$\sum_{i\in I, i\leq l} x_{il} = 1 \qquad \forall\, l \in I \tag{3.28}$$

$$x_{il} \leq x_{ii} \qquad \forall\, i,l \in I \tag{3.29}$$

$$z_{ij} \geq x_{hi} + x_{hl} - 1 \qquad \forall\, \{i,j\} \in \bar{E}, h \in I, \{l,j\} \in E \tag{3.30}$$

$$x_{il}, z_{ij} \in \{0,1\} \qquad \forall\, i,l \in I, j \in J \tag{3.31}$$

Constraint 3.27 ensures that we actually group the sessions in p clusters. Constraint 3.28 ensures that each session has one and only one representative, which can be translated as each session being in exactly one cluster. Constraint 3.29 forces a session that is representative of another one to be representative of itself too. The last one (3.30) forces us to pay an arc, when the right condition is met.

3.4 Fourth model

In this model the clusters are considered as combinatorial objects. Since the number of variables is exponential, a column generation approach is suggested.

Let $T = \{(I_t, J_t)\}$ be the set of every possible cluster, where I_t and J_t are, respectively, the set of left nodes and right nodes belonging to cluster t ($I_t \subseteq I$ and $J_t \subseteq J$). t represents the (transpose) characteristic vector of a cluster, that is $t_i = 1$, if $i \in I_t$, and $t_i = 0$ otherwise. Let $c_t = |(I_t \times J_t) \cap \bar{E}|$ denote the cost of cluster t and $a_{it} = 1$ if $i \in I_t$, $t \in \{1, ..., |T|\}$.

We now define the variable:

$$\lambda_t = \begin{cases} 1 \text{ if we select cluster } t \\ 0 \text{ otherwise} \end{cases}$$

Column Generation model: linear formulation (model-4)

$$\min \quad \sum_{t \in T} c_t \lambda_t \tag{3.32}$$

$$\text{s.t.} \quad \sum_{t \in T} a_{it} \lambda_t = 1 \qquad \forall\, i \in I \tag{3.33}$$

$$\sum_{t \in T} \lambda_t = p \tag{3.34}$$

$$\lambda_t \in \{0, 1\} \qquad \forall\, t \in T \tag{3.35}$$

Constraint 3.33 expresses the partition of the sessions. Constraint 3.34 ensures that the clusters that we take are exactly p.

3.5 Fifth model

This model is based on integer variables. The p clusters are identified by a number from 1 to p. This model suffers from symmetrical solutions. Furthermore it is the only one that cannot be linearized easily: ad-hoc techniques must be used, that reduce this problem to a formulation similar to (model-3) but worse.

$$x_i = \text{ the number of the cluster to which session } i \text{ belongs}$$

$$z_{ij} = \begin{cases} 0 \text{ if we pay the edge } \{i, j\} \\ 1 \text{ otherwise} \end{cases}$$

Note that the meaning of the values that the z variable assumes is the contrary of what is normally used. This is necessary, since the mathematical way to test whether two sessions are in the same cluster is obtained by taking the difference of the relative x variables: if the difference is zero, than they share the same cluster (this can be seen in constraint 3.37).

Linear integer formulation, with a non-linear non-quadratic constraint (model-5)

$$\min \quad \sum_{\{i,j\}\in\bar{E}} (1 - z_{ij}) \tag{3.36}$$

$$\text{s.t.} \quad z_{ij} \leq |x_i - x_l| \qquad \forall \{i,j\} \in \bar{E}, \{l,j\} \in E \tag{3.37}$$

$$x_i \in \{1, ..., p\} \qquad \forall\, i \in I \tag{3.38}$$

$$z_{ij} \in \{0, 1\} \qquad \forall\, i \in I, j \in J \tag{3.39}$$

Constraint 3.37 forces z_{ij} to take value 0 when we must pay the edge $\{i, j\}$. Constraint 3.38 restricts the domain of variable x inside the range of clusters.

3.6 Properties of the models

In this section we list the most important properties of the previous formulations.

Proposition 1. *The optimal solution always uses all the p clusters (monotonicity of the objective function w.r.t. the number of clusters).*

Proof. It is sufficient to prove that adding a cluster to a solution cannot worsen the objective function, but generally makes it better. Thus, by induction, if a solution has $p - 1$ clusters, it is always possible to find a better or equal cost solution with p clusters.

Let $T_k = \langle I_k, J_k, E_k \rangle$ be a cluster containing $\omega = |I_k|$ sessions and having cost c_k. If it is a biclique ($c_k = 0$), then splitting it into two clusters will still generate two clusters with cost 0. Otherwise we can select one node i from I_k that has at least one missing edge (say $\{i, j\}$). If we remove i from T_k and put it into another empty cluster T_r, $c_r = 0$ because T_r contains only one node. The original cluster T_k, now T'_k has a cost c'_k reduced by at least one unit. Thus $c'_k + c_r \leq c_k$. ∎

Proposition 2. *Overlapping clusters (i.e. sharing at least one session) have a cost greater than or equal than disjoint ones.*

Proof. Let $T_k = \langle I_k, J_k, E_k \rangle$ and $T_r = \langle I_r, J_r, E_r \rangle$, with $|I_k| \geq 2$ and $|I_r| \geq 2$, be two clusters that share only one node i ($I_k \cap I_r = i$). If we remove i from one of the two clusters we still obtain two clusters, that are disjoint

and we still cover all the session nodes. Removing a node from a cluster though can only lower the cost of that cluster or keep it the same. Suppose we remove i from T_k, obtaining $T'_k = \langle I'_k, J'_k, E'_k \rangle$, where $I'_k = I_k \backslash i$. The only case in which the cost remains unchanged is when i is connected to all the clients in J_k. Otherwise the cost of T'_k lowers by at least one unit.

The reasoning can be easily extended to the case of sharing more than one node. ∎

Proposition 3. *The continuous relaxation of (model-1a) has an integer solution.*

Proof. Let $D = \langle I, J, E, p \rangle$ be an instance of (model-1a) and assume that $(\hat{x}, \hat{y})$ is a non-integer optimal solution of the continuous relaxation D^R of D. We first observe that for any $\{i, j\} \in E$, if $\hat{x}_i^k > 0$ then we have $\hat{y}_j^k = 1$ since otherwise $\sum_{k \in K} \hat{x}_i^k \cdot \hat{y}_j^k < 1$. We define for any $i \in I$, the set $K(i) = \{k \in K | \hat{x}_i^k > 0\}$ and for any $j \in J$, the set $U(j) = \cup_{\{i,j\} \in E} K(i)$. Now the objective function writes $\sum_{\{i,j\} \in \bar{E}} \sum_{k \in K(i)} x_i^k \cdot y_j^k$.

Let us first consider the vector $(\hat{x}', \hat{y}')$ defined as follows: $\hat{x}' = \hat{x}$ and $y_j'^K = 1$ if $k \in U(j)$ and 0 otherwise. $(\hat{x}', \hat{y}')$, which clearly satisfies 3.2 and 3.3, is a solution of D^R. Moreover, since $\hat{y}' \leq \hat{y}$, $(\hat{x}', \hat{y}')$ is also an optimal solution of D^R.

Assume now that $\hat{x}_i'^{m(i)} = min\{\hat{x}_i'^k \mid k \in K(i)\}$ and consider the vector $(\hat{x}'', \hat{y}'')$ defined as follows: $\hat{y}'' = \hat{y}'$; $\hat{x}_i''^{m(i)} = 1$; $\hat{x}_i''^k = 0$ for $k \in K(i) \backslash \{m(i)\}$. We first observe that for any $\{i_0, j_0\} \in E$, we have $\sum_{k \in K} \hat{x}_{i_0}''^k \cdot \hat{y}_{j_0}''^k = \hat{y}_{j_0}''^{m(i_0)} = 1$ and $\sum_{k \in K} \hat{x}_{i_0}''^k = 1$. So $(\hat{x}'', \hat{y}'')$ is a solution of D. If C'' (respectively C') is the cost of $(\hat{x}'', \hat{y}'')$ (resp. $(\hat{x}', \hat{y}')$), we get from the preceding expression of the objective function that $C'' - C' = \sum_{\{i,j\} \in \bar{E}} \sum_{k \in K(i)} \hat{y}_j''^k (\hat{x}_i''^k - \hat{x}_i'^k)$. Since from the definition of $m(i)$, each term of the sum is non-positive, we conclude that $C'' - C' \leq 0$. Thus, $(\hat{x}'', \hat{y}'')$ is an optimal solution of D. ∎

Proposition 4. *The relaxed version of (model-1d), when the variables y_j^k and z_{ij}^k are continuous over [0 1], has an integer optimum.*

Proof. This result can be easily derived by considering the problem obtained when the x_i^k variables are fixed to integer values (either 0 or 1). Constraint 3.14 forces y_j^k to be 1 whenever x_i^k is 1. Otherwise, when x_i^k is 0, y_j^k will be 0 too for the minimization. In constraint 3.20 variable z is set to 1 when both x and y have value one. For the previous considerations, the only left value for x and y other than 1 is 0. When either one of them

assume value 0, the sum at the right of the inequality is ≤ 0 and z will be set to 0 in this case, because of the objective function. ∎

A similar property holds for (model-1b), (model-2) and (model-3) whenever variables z are continuous over [0 1]. All the aforementioned models, in fact, share the same linear constraint expressing z as a product of two variables. The proof is easily derived, as this is a special case of Proposition 4.

Proposition 5. *The relaxed version of (model-5), when the variables z_{ij} are continuous over [0 1], has an integer optimum.*

Proof. The result is similar to Proposition 4. If the x variables are kept integer, the modulo of their difference in constraint 3.37 is integer too (taking values in $\{0, 1, ..., p-1\}$). When the difference is 0 z is forced to 0 too, otherwise z can assume any continuous value in $[0, 1]$. Anyway, because of the objective function, it is convenient to set it to 1. So the optimal solution will always assign either 0 or 1 to variables z. ∎

Chapter 4

Solution methods

The main goal of this work is to develop effective methods for the solution of the k-CmBCP. We designed and implemented three approaches:

1. A Local Search heuristic.
2. A Column Generation procedure.
3. A Branch-and-Price method.

We introduce the basic theory of a *Local Search* algorithm first. Then we explain the architecture of the heuristic. Since the focus of the problem is the clustering of left shore nodes, we refer to them simply as nodes. When confusion may arise between the two shores of the bipartite graph, we use the terms left nodes and right nodes.

4.1 Introduction to heuristic methods

Heuristics are methods for solving an optimization problem. Their peculiarity is that they fail to ensure the optimal solution (if it exists), but they are generally very efficient, yielding a "good" solution in a relatively short amount of time. The previous term "good" usually refers to the objective function, but the quality of a solution changes a lot based on the problem and the chosen evaluation method.

Given that many of the practical problems that are encountered are NP-hard, it is not surprising that heuristic algorithms play an important role in solving discrete optimization problems. *Meta-heuristics* are general frameworks to build heuristics. A broad survey about the best-known of them can be found in the books [15, 16].

Heuristics are classified in:

1. **Constructive algorithms**: they construct a solution by means of a sequence of operations that expand a partial solution.
2. **Local search algorithms**: they search for a solution of high quality in the solution space.

The heuristic we have implemented is mixed: it is based on local search, but it contains constructive methods to generate starting solutions.

4.1.1 Greedy method

A Greedy method builds a solution by sequentially expanding a partial one. Its time complexity is generally linear with the size of the instance, because it iteratively builds the solution without ever going back to choices made previously.

To design a Greedy method, we need to define the *ground set* E, that is the set of elements composing the solution of the problem, and the *family of independent sets* F, that is the set of all partial solutions being feasible for the problem.

Algorithm 1 The general Greedy algorithm.

```
1: procedure GREEDY
2:     s ← ∅
3:     repeat
4:         s ← best(E)
5:         E ← E \ {e}
6:         if feasible(s, e) then
7:             s ← s ∪ {e}
8:         end if
9:     until E = ∅
10: end procedure
```

The algorithm builds the set $s \in F$ (the solution of the problem), starting from the empty set and, at each step, inserting in it the most "promising" element of E. It iteratively chooses the best element e remaining in E, removes it from E and put it into the solution s that it is constructing. e is incorporated in s only if that does not violate the constraints of the problem (i.e. $s \cup \{e\} \in F$). The cycle is repeated until E becomes empty.

Algorithm 1 shows what we have just explained. Procedure `best`(E) chooses the best element from E according to a pre-set criterion, that must be developed ad-hoc for the particular problem. It may be that several criteria are possible, yielding a different solution each. Procedure `feasible`(s, e) is a logical function:

$$\text{feasible}(s, e) = \begin{cases} true, & \text{if } s \cup \{e\} \in F \\ false, & \text{otherwise} \end{cases}$$

Greedy methods are usually extremely fast, but yield the same solution at every execution. A way to avoid this defect is to render the choice of the best element non-deterministic, so that if the algorithm is called more than once, at each execution - at least in principle - different choices are made and alternative solutions are generated. This idea forms the basis of *GRASP* (Greedy Randomly Adaptive Search Procedure). Its steps are shown in Algorithm 2, where c_j indicates the cost of solution j. The procedure `best`(E), in this case, takes into consideration all the elements that, if inserted in s, would worsen the objective no more than $\alpha\%$ with respect to the best one. It then randomly selects one of these elements.

Algorithm 2 The function `best`(E) in a GRASP algorithm.

```
1: function BESTGRASP(E, α)
2:     v ← min{c_j | j ∈ E}
3:     E' ← {j | c_j ≤ v(1 + α)}
4:     return (random(E'))
5: end function
```

4.1.2 Local Search

Local Search is based on what is perhaps the oldest optimization method: trial and error.

Given an instance (F, c) of an optimization problem, where F is the feasible set and c is the cost mapping, we choose a neighborhood

$$N : F \longrightarrow 2^F$$

which is searched at point $t \in F$ for improvements by the subroutine

$$\text{improve(t)} = \begin{cases} \text{any } s \in N(t) \text{ with } c(s) < c(t) \text{ if such an } s \text{ exists} \\ \text{no otherwise} \end{cases}$$

The general local search algorithm is shown in Algorithm 3. We start at some initial feasible solution $s \in F$ and use subroutine `improve` to search for a better solution in its neighborhood. So long as an improved solution exists, we adopt it and repeat the neighborhood search from the new solution; when we reach a local optimum we stop.

Algorithm 3 The general local search algorithm.

```
1: procedure Local Search
2:     s ← some initial starting point in F
3:     while improve(s) ≠ 'no' do
4:         s ← improve(s)
5:     end while
6:     return s
7: end procedure
```

To apply this approach to a particular problem, we must make a number of choices. First, we must decide how to obtain an initial feasible solution. It is sometimes practical to execute local search from several different starting points and to choose the best result. In such cases, we must also decide how many starting points to try and how to distribute them.

Next, we must choose a "good" neighborhood for the problem at hand and a method for searching it. One can see a clear trade-off here, however, between small and large neighborhoods. A larger neighborhood would seem to hold promise of providing better local optima but will take longer to search, so we may expect that fewer of them can be found in a fixed amount of computer time.

We now give a list of definitions that are useful later, to better describe the strategies that we implemented.

A **combinatorial optimization problem** ([17]) is specified by a *ground set* E, a *solution space* S ($S \subseteq 2^E$) containing a finite number of solutions and a *cost function* f, that is a mapping $f : S \to R$ that assigns a real value to each solution in S called the cost of the solution. Given an instance of the problem, the objective is to find a solution $s^* \in S$ that is **globally optimal**. For a minimization problem, this means that $f(s^*) \leq f(s)$ has to hold $\forall s \in S$. If no confusion can arise, a globally optimal solution is simply called optimal.

For an instance (S, f) of a combinatorial optimization problem, a **neighborhood function** is a mapping $N : S \to 2^S$, where 2^S denotes the power

set $\{V \mid V \subseteq S\}$. The neighborhood function specifies for each solution $s \in S$ a set $N(s) \subseteq S$, which is called the *neighborhood* of s. The cardinality of $N(s)$ is called the *neighborhood size* of s. We say that solution s' is a neighbor of s if $s' \in N(s)$. The neighborhood function N is said to be symmetric in the case that we have $s' \in N(s)$ if and only if $s \in N(s')$.

A solution $\hat{s} \in S$ is called **locally optimal** with respect to N or N-optimal if and only if $f(\hat{s}) \leq f(s)$, $\forall s \in N(\hat{s})$.

Let N_1 and N_2 be two different neighborhood functions for the same instance (s, f) of a combinatorial optimization problem. If for all solutions $s \in S$ we have $N_1(s) \subseteq N_2(s)$, then we say that N_2 **dominates** N_1.

A neighborhood function is called **exact** if each local optimum is also a global optimum.

The **neighborhood graph** of an instance (S, f) of a combinatorial optimization problem and an accompanying neighborhood function N is a directed node-weighted graph $G = (V, E)$. The node set V is given by the set S of solutions, and the arc set E is defined such that $(i, j) \in E$ if and only if $j \in N(i)$. Furthermore, we define the weight of a node as the cost of the corresponding solution. If the neighborhood function is *symmetric*, then the directed graph can be simplified to an undirected graph by replacing arcs (i, j) and (j, i) by a single edge $\{i, j\}$.

The **transition graph** T of an instance of a combinatorial optimization problem and an accompanying neighborhood function is a directed, acyclic subgraph of their neighborhood graph G. It is obtained from G by deleting all arcs (i, j) for which it holds that the cost of solution j is worse than or equal to the cost of solution i. Figure 4.1 shows a symmetric neighborhood graph, where the cost of the solution is indicated inside the node, and the corresponding transition graph.

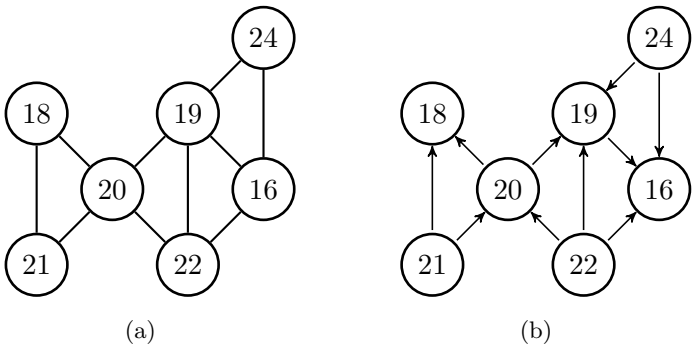


Figure 4.1: Neighborhood graph (a) and its relative transition graph (b).

Executing a Local Search algorithm with *neighbor function* N corresponds to perform a walk on the *neighborhood graph* G. Local Search algorithms use two different strategies to walk on the graph:

1. **Iterative Improvement**: move to a neighbor solution with better cost, hence they stop on local optimum. They only move following the arcs of the transition graph. They use a pivoting rule to distinguish multiple neighbors:
   (a) *First Improvement*: accept the first improving solution
   (b) *Best Improvement*: select the best improving solution
2. **Meta-heuristics**: allow moves to a non-improving solution. They (basically) are classified into:
   (a) *Single walk*: they have mechanism for avoiding local optimum, i.e., accepting non-improving solutions, such as for instance Simulated Annealing and Tabu Search.
   (b) *Multiple walks*: they prevent local optimum by repeating multiple walks, such as for instance, Random Restart, Iterative Local Search, and Genetic Local Search.

4.1.3 Tabu Search

A variation of the local search paradigm that has proved to be versatile and efficient is the *Tabu Search* ([18]). For many problems this basic variation is the only addition needed to find very good solutions.

In the basic local search we generally stop when no improving solution can be found. This method only allows descending moves towards the first local minimum encountered in the solution space. To improve it, we can decide to continually move, even after a local plateau. The variation is to move from solution s to $s' \in N(s)$, where s gives the smallest increase in $f(s)$. However, there is a major drawback: *cycling*. In fact, if s is a local minimum and it is the best solution in $N(s')$, the search will indefinitely move back and forth between s and s'.

To avoid this, the Tabu Search keeps a tabu list T of the recently performed moves (short-term memory), that become forbidden. It behaves as if it were removing elements from $N(s)$. Whenever the local search moves inside $N(s)$ to find a neighbor solution, it cannot select any s' that requires

a move present in the tabu list to reach it. Of course, after a finite interval, a solution must cease to be tabu: without this feature, all the solutions of a neighborhood could become forbidden, freezing the whole search procedure.

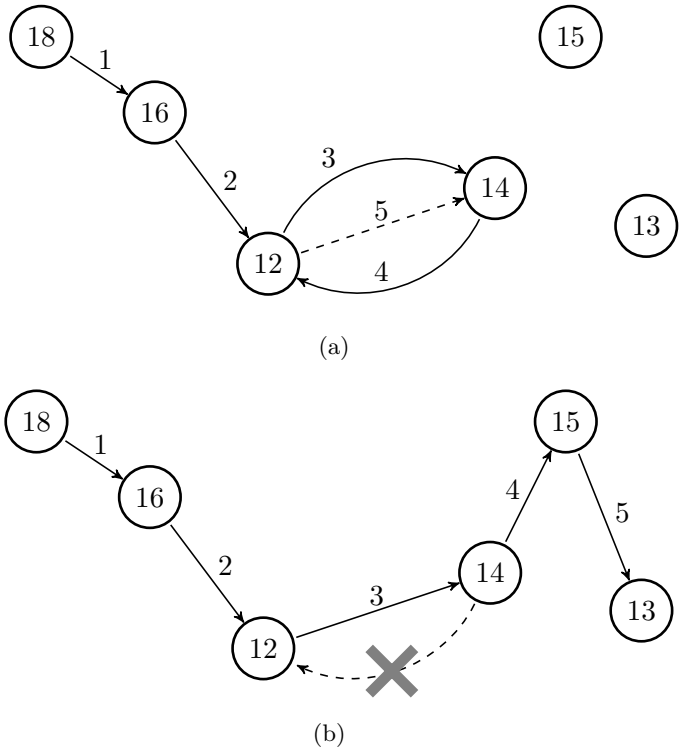


Figure 4.2: Local Search (a) vs Tabu Search (b).

In Figure 4.2 we show two walks on the neighborhood graph. The one on top (a) follows a simple local search approach that always moves to the neighbor that gives the smallest increase in $f(s)$. After a local minimum with value 12 is reached, the search moves to the best solution in its neighborhood, that is 14. However, the best solution in the neighborhood of 14 is the previous 12, so the search goes back. The last two steps can continue indefinitely. Below we show how the same neighborhood graph would be visited by a Tabu Search (b): cycling is avoided, as moving back to solution 12 is forbidden. Going back to 12 will be allowed again when the short-term memory associated with that move will elapse.

With the tabu list, the search has the opportunity to visit solutions that are less promising than others, but that have not been chosen in a long time. Usually Tabu Search presents two important strategies:

1. *Intensification*: to explore deeper promising regions of the solution space.
2. *Diversification*: to explore several different regions of the solution space.

Finally, often the *Aspiration criterion* is implemented too: a tabu move is nevertheless executed if it improves the best solution found so far.

Since the tabu search is a well known search method, it will be used to compare the performance of our heuristic.

4.1.4 Properties of a meta-heuristic

We list here some of the desirable properties that a meta-heuristic should satisfy. In developing our solution method we tried to adhere to them as much as possible. These properties are presented in [16], with a few extensions introduced in [19].

1. *Simplicity*: it should be based on a simple and clear principle, which should be widely applicable;
2. *Precision*: the steps of the meta-heuristic should be formulated in precise mathematical terms, independent of possible interpretations or initial inspiration;
3. *Efficiency*: heuristics for particular problems should provide optimal or near-optimal solutions for either all or, at least, most realistic instances. Preferably, they should find optimal solutions for most problems of benchmarks for which such solutions are known, when available;
4. *Robustness*: its performance should be consistent over a variety of instances, i.e., not merely fine-tuned to some training set and less good elsewhere;
5. *User-friendliness*: it should be clearly expressed, easy to understand and, most importantly, easy to use. This implies there should be as few parameters as possible, ideally none;
6. *Generality*: it should lead to good results for a wide variety of problems;

4.2 Heuristic for the *k*-CmBCP

In this section we shall present the algorithm used to solve the k-CmBCP. As stated previously it is based on Local Search, but it incorporates *Greedy* methods.

First let us briefly explain the structure of the algorithm. At the beginning a starting solution has to be generated. Several approaches are investigated: random solutions, construction with a Greedy or a GRASP method. Then a local search is applied to the starting solution. Different strategies can be applied, both for the descent phase and the escape from valleys. Whatever method is used, a feasible solution is returned at the end, hopefully very close to the optimal one (ideally the optimal solution).

We can now go into further details for each part of the algorithm. Note that in the following sections we are going provide two abstraction levels of the algorithm:

1. the theoretical concepts characterizing the heuristic;
2. how these concepts are applied to the specific problem.

For each feature of the heuristic, we shall first describe it in general terms, i.e. in the form that can be applied to any optimization problem. Right after that, we shall describe how that feature is specialized to suit our problem. Whenever more than one choice is available, we also explain what we have chosen and why.

4.2.1 Starting solutions

A starting solution must be generated in order to even start a Local Search. Three approaches have been implemented. It is important that the method used is fast to execute and easy to implement, because, usually, the choice of a starting solution does not affect the performance of the search so much.

Random generation

The simplest and fastest way is to generate a completely random solution. To do that, we assign each session to a random cluster among the p available. Some clusters can be empty but that is not a concern, since every minimization technique will quickly fill them up.

Greedy/GRASP

Both Greedy and GRASP methods have been implemented.

In the Greedy approach, all nodes $i \in I$ are considered sequentially. Each i is inserted in a cluster $k \in K$, so that the added cost is minimum. Thus, for each node i, if we denote with s_i^k the solution obtained by inserting i into cluster k, the procedure $\mathbf{best}(K)$ behaves this way:

$$\mathbf{best}(K) = arg\min_{k \in K}\{f(s_i^k)\}$$

In the GRASP approach, the same structure is used, but this time, we select a random $k \in K$ among those which worsen the objective function no more than $\alpha\%$ with respect to the best one. The value α can assume any reasonable value: the majority of our tests have been executed with $\alpha = 5$.

4.2.2 Variable Neighborhood Search

Once the starting solution has been generated, a Local Search algorithm is applied to it. The first method we are going to describe is an application of *Variable Neighborhood Search* (VNS), a meta-heuristic which systematically exploits the idea of neighborhood change, both in descent to local minima and in escape from the valleys which contains them ([20, 19]). VNS heavily relies upon the following observations:

1. A local minimum with respect to one neighborhood structure is not necessarily a local minimum for another neighborhood structure.
2. A global minimum is a local minimum with respect to all possible neighborhood structures.
3. For many problems, local minima with respect to one or several neighborhoods are relatively close to each other.

This last observation is empirical. It implies that a local optimum often provides some information about the global optimum.

Unlike many other meta-heuristics, the basic schemes of VNS and its extensions are simple and require few or, sometimes, no parameters. Therefore, in addition to providing very good solutions, often in simpler ways than other methods, VNS gives insight into the reasons for such a performance, which, in turn, can lead to more efficient and sophisticated implementations. This idea has its predecessor in the *variable metric method*, suggested in [21, 22].

We shall first describe the functions that characterize the moves of the heuristic. Let us denote with N_k , $(k = 1, ..., k_{max})$, a finite set of pre-selected neighborhood structures, and with $N_k(s)$ the set of solutions in the k-th neighborhood of s. The index k indicates the size of the neighborhood. We will also use the notation $N'_k, k = 1, ..., k'_{max}$, when describing local descent. N'_k can be different from N_k, but even for them a bigger k stands for a bigger size. We call $s' \in S$ a local minimum with respect to N_k, if there is no solution $s \in N_k(s) \subseteq S$ such that $f(s) < f(s)$. The neighborhood structure $N(s)$, with no index, is the simplest and smallest possible, generally the one used in traditional local search methods: it consists of all possible solutions obtained from s by selecting a node and moving it into another cluster. To simplify the description of the algorithms we always use t_{max} to indicate the stopping condition.

In Algorithm 4 is summarized the most used descent method in local search, `BestImprovement`, sometimes called *steepest descent*. An initial solution s is given and the output consists of a local minimum. As the steepest descent may be time-consuming, an alternative is to use the first descent heuristic (`FirstImprovement`). Solutions $s_i \in N(s)$ are then enumerated systematically and a move is made as soon as a direction for the descent is found. This is summarized in Algorithm 5.

Algorithm 4 Best Improvement heuristic.

```
1: procedure BestImprovement(s)
2:     repeat
3:         s' ← s
4:         s ← arg min_{t∈N(s)} f(t)
5:     until f(s) ≥ f(s')
6: end procedure
```

The difference of overall performance between first and best improvement is not easy to foresee as it depends on the particular problem or algorithm in which they are used. Experimental tests should dictate which one is better; for an investigation see [23].

Function `NeighborhoodChange`, shown in Algorithm 6, compares the new value $f(s')$ with the incumbent value $f(x)$ obtained in the neighborhood k (line 2). If an improvement is obtained, k is returned to its initial value and the new incumbent updated (line 3 and 4). Otherwise, the next neighborhood is considered (line 5).

Algorithm 5 First Improvement heuristic.

```
1: procedure FirstImprovement(s)
2:     repeat
3:         s' ← s
4:         i ← 0
5:         repeat
6:             i ← i + 1
7:             s ← arg min{f(s), f(s_i)}, s_i ∈ N(s)
8:         until f(s) = f(s_i) or i = |N(s')|
9:     until f(s) ≥ f(s')
10: end procedure
```

Algorithm 6 Neighborhood change or "Move or not" function.

```
1: procedure NeighborhoodChange(s, s', k)
2:     if f(s') < f(s) then
3:         s ← s'
4:         k ← 1                                  // Make a move
5:     else
6:         k ← k + 1                              // Next neighborhood
7:     end if
8: end procedure
```

Algorithm 7 Random selection within a neighborhood.

1: **procedure** SHAKE(s, k)
2: $s' \leftarrow$ `select`$(t \mid t \in N_k(s))$
3: **return** s'
4: **end procedure**

The *Variable Neighborhood Descent* (VND) method is obtained if the change of neighborhood is performed in a deterministic way. Its steps are presented in Algorithm 8.

Algorithm 8 Steps of the basic VND.

1: **procedure** VND(s, k'_{max})
2: **repeat**
3: $k \leftarrow 1$
4: **repeat**
5: $s' \leftarrow arg\min_{t \in N'_k(s)} f(t)$
6: NEIGHBORHOODCHANGE(s, s', k)
7: **until** $k > k'_{max}$
8: **until** no improvement is obtained
9: **end procedure**

In the descent phase, very few neighborhoods should be used: generally one or two. We set $k'_{max} \leq 3$. Note that the final solution should be a local minimum with respect to all k'_{max} neighborhoods; hence the chances to reach a global one are larger when using VND than with a single neighborhood structure.

The last function used is `Shake`. It is shown in Algorithm 7 and it extracts a random solution s' from the k-th neighborhood of s ($s' \in N_k(s)$).

The previous functions are combined to obtain the final VNS algorithm. A first version is called *Basic* VNS (VNS), that combines deterministic and stochastic moves. Its steps are given in Algorithm 9.

Solution s' is generated at random in line 5 in order to avoid cycling, which might occur if a deterministic rule were applied. In line 6 the `FirstImprovement` local search is usually adopted. However, it can be replaced with `BestImprovement`. A graphical representation of the behavior of the VNS is shown in Figure 4.3. When a local minimum is reached, the size of the neighborhood is iteratively increased in the attempt of escaping

Algorithm 9 Steps of the basic VNS.

```
1: procedure VNS(s, k_max, t_max)
2:     repeat
3:         k ← 1
4:         repeat
5:             s' ← Shake(s, k)
6:             s'' ← FirstImprovement(s')
7:             NeighborhoodChange(s, s'', k)
8:         until k > k_max
9:         t ← CpuTime()
10:    until t > t_max
11: end procedure
```

from the valley. As soon that a new minimum is found, the size of the neighborhood is reset to the initial value and the search continues. In this way larger and slower neighborhoods are used only when really needed. Thus VNS performs many short steps rather than a few big ones.

The last variation we implemented consists in replacing the local search in line 6 by VND. This algorithm is called *General* VNS (GVNS) and has led to many successful applications (see, for example, [24, 25]). Its steps are given in Algorithm 10.

Algorithm 10 Steps of the general VNS.

```
1: procedure GVNS(s, k'_max, k_max, t_max)
2:     repeat
3:         k ← 1
4:         repeat
5:             s' ← Shake(s, k)
6:             s'' ← VND(s', k'_max)
7:             NeighborhoodChange(s, s'', k)
8:         until k > k_max
9:         t ← CpuTime()
10:    until t > t_max
11: end procedure
```

We now describe the neighborhood structures we used. We already defined $N(s)$ as the smallest one. In the VND method three neighborhoods

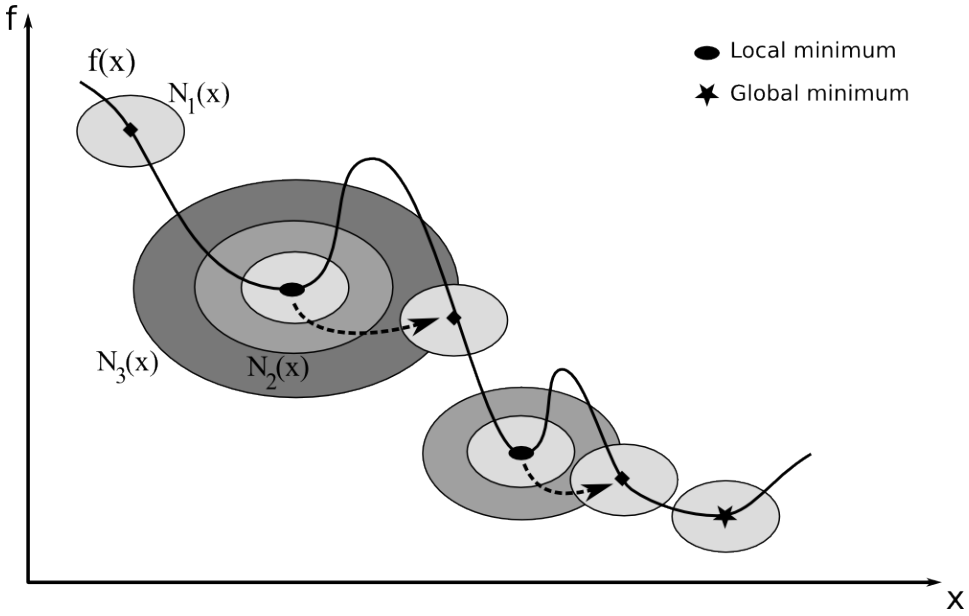


Figure 4.3: Basic VNS.

N'_k were used.

Neighborhood 1 (N'_1) is the same used in simple search: select one node and move it to another cluster.

Neighborhood 2 (N'_2) consists in swapping two nodes belonging to different clusters. Consider node i and j, belonging to clusters k and l respectively. The move associated with these two nodes would put i in l and j in k.

Neighborhood 3 (N'_3) selects two distinct nodes and put each of them into a cluster different from their original one. Consider node i_1 and i_2, belonging to clusters k_1 and k_2 respectively. The move associated with these two nodes would put i_1 in k_3 and i_2 in k_4, with $k_3 \neq k_1$ and $k_4 \neq k_2$ (k_3 could be equal to k_4).

To understand how these three neighborhoods are related in the solution space, consider Figure 4.4. Take any solution s. The smallest neighborhood is $N'_1(s)$. $N'_2(s)$ has bigger size and yet it does not share any solution with $N'_1(s)$. It is not possible, in fact, to move a single node into another cluster by performing a swap between two nodes. Whereas, a swap move can only be obtained by two consecutive single moves. The two neighborhoods are

disjoint sets. This characteristic is a consequence of the particular move that defines $N'_2(s)$. We'd like to point out that this is a desirable property of different neighborhoods, especially in VND. Consider that the search is deterministic in this case, so switching to another neighborhood during the search is good only if a different space is explored. Neighborhood is changed only after a local minimum is found. The best condition is clearly to completely change space, since the previous one has been evaluated in its totality.

Finally $N'_3(s)$ has the highest cardinality of the three. Moving two nodes allows to reach any solution in $N'_1(s)$ an $N'_2(s)$. However, for the same reason we stated before, we formulated it in such a way that $N'_3(s)$ does not contain any solution in $N'_1(s)$ nor in $N'_2(s)$. It is sufficient to select two *distinct* nodes and moving them to two clusters, *both different from the original ones.* Thus VND uses three disjoint sets during its execution:

$$N'_i(s) \cap N'_j(s) = \emptyset,\ \forall i, j \in \{1, \ldots, k'_{max} \mid i \neq j\}$$

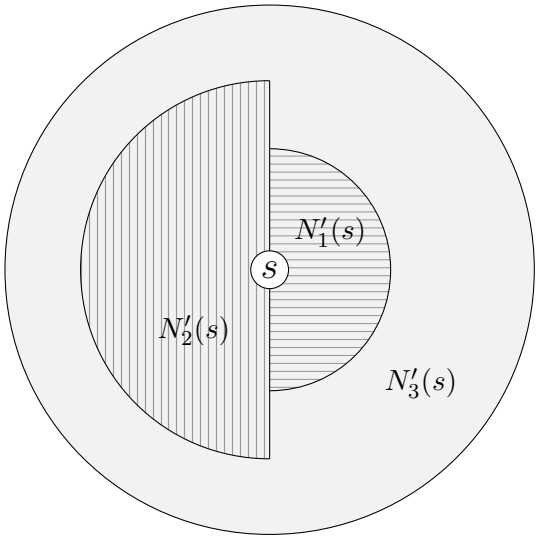


Figure 4.4: Representation of the three neighborhoods of the VND.

In the VNS methods instead, k is not limited to three and it determines how many nodes can be moved into another cluster. It can be easily derived that the maximum value that k can take is n: in such a case N_k becomes the whole solution space. So neighborhood $N_1(s)$ is composed of all the solutions obtained from s by moving one node to another cluster, $N_2(s)$ is composed of all the solutions obtained from s by moving two distinct nodes

to two clusters different from their original ones, and so on. Note that $N_2 = N_3'$ and $N_1 = N_1' = N$, that is the simplest neighborhood structure is used in every search as the first choice (in the `FirstImprovement` and `BestImprovement` functions it is the only neighborhood used). The size of a neighborhood N_k is:

$$|N_k| = \prod_{i=1}^{k}(n - i + 1)(p - 1)$$

The second neighborhood used in VND (N_2') has a different structure and its size does not follow the previous expression. The actual size depends on how the clusters are organized in the solution s, however we can provide an upper bound:

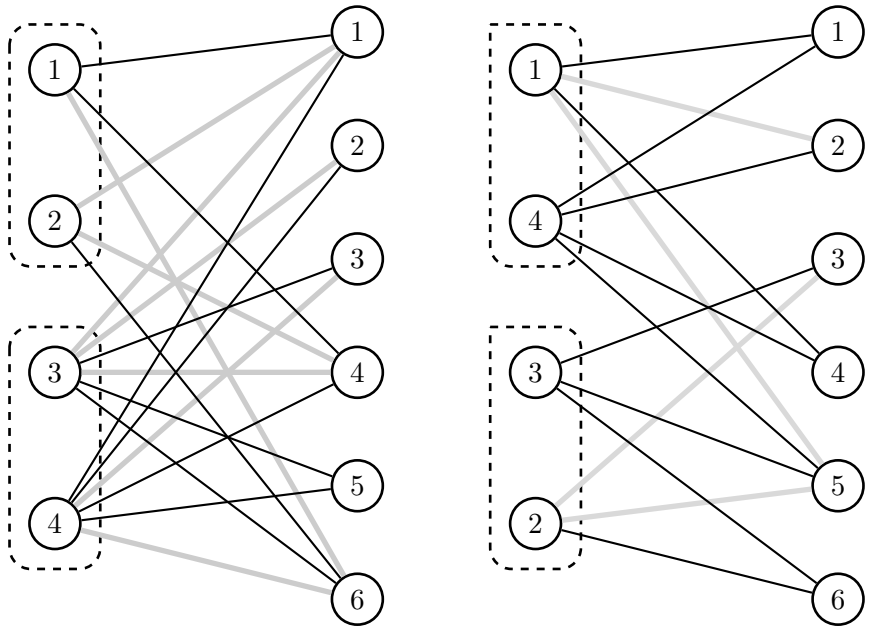
$$|N_2'| < \frac{n^2}{2}$$

In Figure 4.5 we show a situation in which the change of neighborhood performed by VND allows to escape from a local minimum: in this particular case a global minimum is found too. In the figure $n = 4$, $m = 6$ and $p = 2$. The solution s on the left is a local minimum with respect to $N_1'(s)$, for which no further move is possible. The cost of the solution is 8 and it can be easily verified that every neighbor solution in $N_1'(s)$ has cost greater or equal than 8. Since a move in $N_1'(s)$ consists in moving a node into another cluster, its size in this instance is $n(p-1) = 4 \times 1 = 4$. By enumerating all the possible combinations we get:

Node moved	New cost
1	9
2	10
3	9
4	8

Solution s, however, does not represent a local minimum for $N_2'(s)$. In fact, if we move inside $N_2'(s)$ we can obtain the solution on the right, the global optimum, in one step: we need to swap nodes 2 and 4 and the new total cost becomes 4.

Example We illustrate the basic steps of VNS on the instance shown in Figure 4.6, where $n = 6$, $m = 8$ and $p = 2$. On the right a random starting



Total cost: 8 Total cost: 4

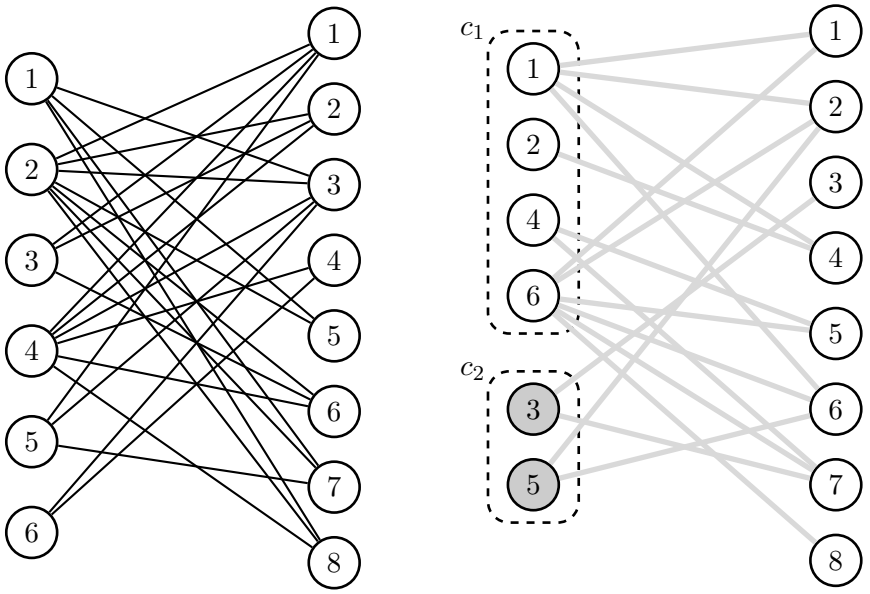
(a) (b)

Figure 4.5: Global optimum (b) obtained by a swap move from a local optimum (a).

solution is shown, with cost = 17. For clarity only the edges that we pay are shown.

Figure 4.7 shows the steps of the algorithm: each solution obtained is represented in a box and the sequential number of the steps is written at the bottom left of each box. Given that the number of edges is quite big, to make things more clear we shall omit the right shore nodes: only the total cost of a solution will be written at the bottom right of each box. The nodes need to be grouped into two clusters that we shall call c_1 and c_2. We use `FirstImprovement` as the local search descent method, indicated as `LS`(s) in the figure, and $k_{max} = 2$ (the value is quite low, but it is necessary to keep the example short).

We start at random solution s_0, with $c_1 = \{1, 2, 4, 6\}$, $c_2 = \{3, 5\}$ and cost = 17. From there we perform a `Shake`$(s_0, 1)$, obtaining s_1, with $c_1 = \{1, 2, 4, 5, 6\}$, $c_2 = \{3\}$ and cost = 18. `LS`(s_1) ends in solution s_2, which is the same one we started with. Since no improvement has been achieved,



$n = 6$, $m = 8$, $p = 2$ Starting solution (cost = 17)

(a) (b)

Figure 4.6: An instance of the k-CmBCP, with a starting solution (b).

k is increased by 1. `Shake`$(s_2, 2)$ returns solution s_3, with $c_1 = \{1, 3, 5, 6\}$, $c_2 = \{2, 4\}$ and cost = 23. `LS`(s_3) ends in solution s_4, with $c_1 = \{1, 2, 3, 4\}$, $c_2 = \{5, 6\}$ and cost = 15. Last move improved the solution, so s_4 is kept and k is reset back to 1. We now call `Shake`$(s_4, 1)$, obtaining s_5, with $c_1 = \{1, 2, 3, 4, 6\}$, $c_2 = \{5\}$ and cost = 18. `LS`(s_5) ends in solution s_6, with $c_1 = \{1, 2, 5\}$, $c_2 = \{3, 4, 6\}$ and cost = 14. Last move improved the solution, so s_6 is kept and k is reset back to 1. s_6 is the global optimum too. VNS performs four other steps before terminating, as no improving solution can be found in neither one of the neighborhoods. The procedure return s_6 as best solution found.

Decomposition Search

This improvement of the basic VNS is particularly suited for solving large instances. Two variations have been implemented.

The first one is presented in [26] and is called Variable Neighborhood Decomposition Search (VNDS). It extends the basic VNS into a two-level VNS scheme based upon the decomposition of the problem. Its steps are presented in Algorithm, where t_d is an additional parameter and represents

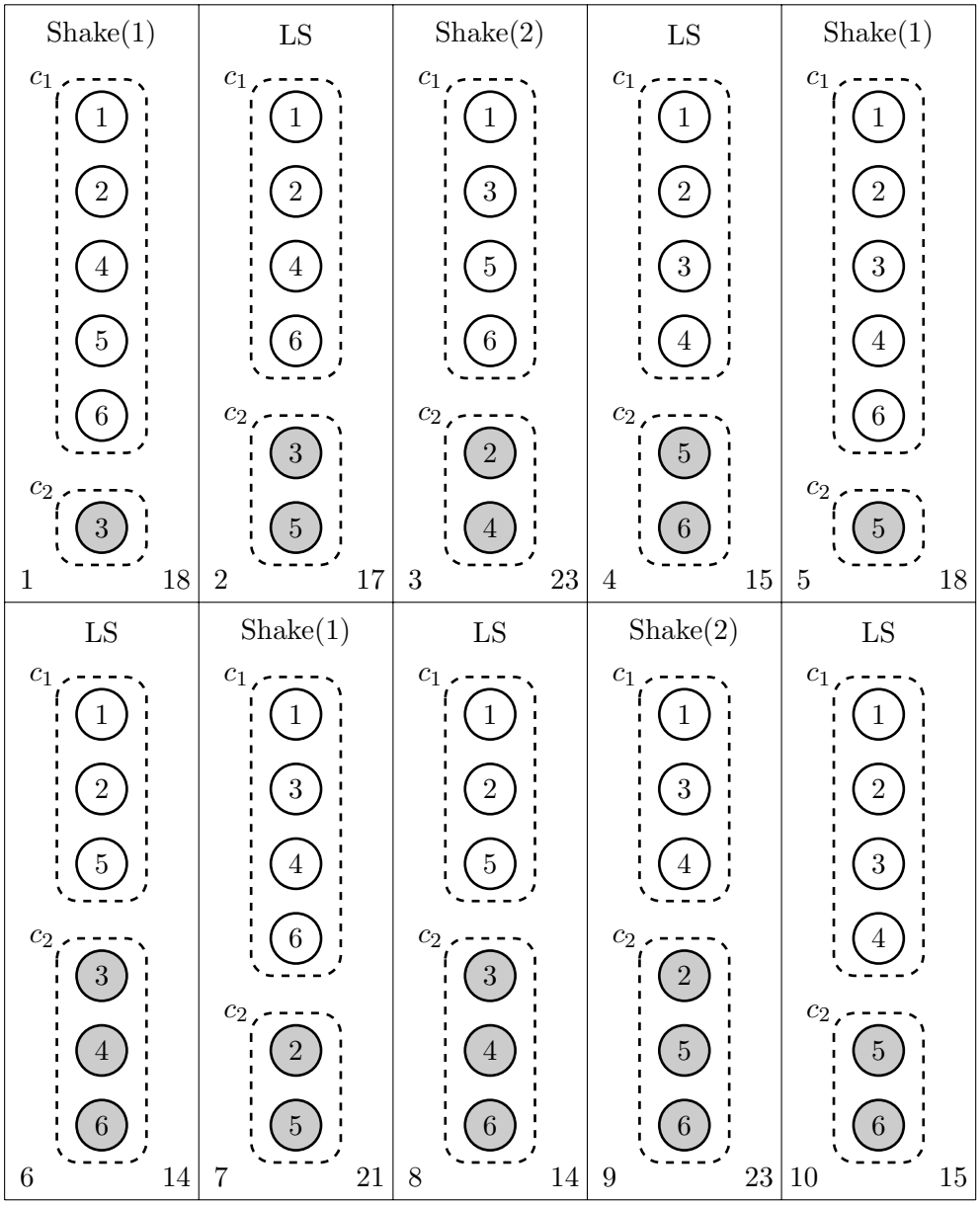


Figure 4.7: Steps of the BVNS for solving the k-CmBCP.

the running time given for solving decomposed (smaller-sized) problems by VNS.

For ease of presentation, but without loss of generality, we assume that the solution s represents the set of some elements. In line 6 we denote with y a set of k solution attributes present in s' but not in s ($y = s' \setminus s$). In line 7 we find the local optimum y' in the space of y; then we denote with s'' the corresponding solution in the whole space S ($s'' = (s' \setminus y) \cap y'$). We notice that exploiting some boundary effects in a new solution can significantly improve its quality. This is why, in line 9, we find the local optimum x''' in the whole space S using s'' as an initial solution. If this becomes time-consuming, then at least a few local search iterations should be performed.

Algorithm 11 Steps of VNDS, first approach.

```
1: procedure VNDS(s, k_max, t_max, t_d)
2:     repeat
3:         k ← 2
4:         repeat
5:             s' ← Shake(s, k)
6:             y ← s' \ s
7:             y' ← VNS(y, k, t_d)
8:             s* ← (s' \ y) ∪ y'
9:             s'' ← FirstImprovement(s*)
10:            NeighborhoodChange(s, s'', k)
11:        until k > k_max
12:        t ← CpuTime()
13:    until t > t_max
14: end procedure
```

The second is a Decomposition Search that uses a more deterministic approach, while the previous one had a stochastic component. The steps of this variation are shown in Algorithm 12.

The idea is to find a "promising" part y of the current solution s and apply a descent method to the new reduced problem obtained by considering y only. This time we let the VNS reach a local minimum. Then we call the procedure `extractPart`(s), that defines which part of the solution to choose and how the subproblem is composed. Again, the search in the reduced problem is iterated until a limit t_d is expired. Note that the k_{max} that is passed to the VNS is exactly the size of y: this is aimed at exploring

Algorithm 12 Steps of VNDS, second approach.

```
1: procedure VNDS(s, k_max, t_max, t_d)
2:     repeat
3:         k ← 1
4:         repeat
5:             s' ← SHAKE(s, k)
6:             s'' ← FIRSTIMPROVEMENT(s')
7:             y ← extractPart(s'')
8:             y' ← VNS(y, |y|, t_d)
9:             s* ← (s'' \ y) ∪ y'
10:            NEIGHBORHOODCHANGE(s, s*, k)
11:        until k > k_max
12:        t ← CpuTime()
13:    until t > t_max
14: end procedure
```

virtually all the subproblem space. Then the solution is recomposed. At this point another descent method would be rather useless. s'' is already a local minimum and the decomposed search tries to make the total cost even lower by focusing on a smaller portion of it. Improvements cannot be found with simple moves.

The criterion we adopted in `extractPart` is specific to our problem. Consider the problem instance (I, J, p), with I being the set of left nodes, J being the set of right nodes, and p the number of clusters. We select two clusters k_1 and k_2 in our current solution $s^\star$. Let us indicate with I_1 and I_2 the set of left nodes contained in k_1 and k_2, respectively. Similarly, J_1 and J_2 are, respectively, the set of right nodes contained in k_1 and k_2. We then perform a local search in the subproblem $(I_1 \cup I_2, J_1 \cup J_2, 2) = (I', J', 2)$. The subproblem is a lot easier than the original one, especially if $p >> 2$, since it consists of moving the nodes from one cluster to the other one. Computing the objective function is faster too, because the number of right nodes is generally lower than the original instance. Without counting the symmetries, the new solution space S' has size:

$$|S'| = 2^{|I'|-1}$$

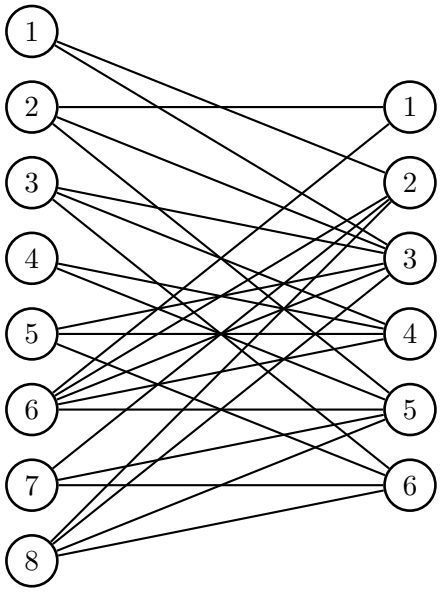
The metric we used to define the most promising pair of clusters is aimed at making the following search useful: if we try to recombine the nodes

between two clusters, it is reasonable to do it with the clusters that share a lot of right nodes. Let us remind that left nodes cannot be shared, because there is a partition constraint on their set I. Given k_1 and k_2, we define $\Gamma_{(1,2)}$ as the set of right nodes shared by k_1 and k_2 ($\Gamma_{(1,2)} = \{j \mid j \in (J_1 \cap J_2)\}$), with $\gamma_{(1,2)}$ being the cardinality of $\Gamma_{(1,2)}$ ($\gamma_{(1,2)} = |\Gamma_{(1,2)}|$). γ is already an interesting metric to consider, but it is absolute and it does not take into account the size of k_1 and k_2. So we adopted a relative value α to select the most promising pair of clusters:

$$\alpha_{(1,2)} = \frac{\gamma_{(1,2)}}{|J_1| + |J_2|}$$

As stated before, this improvement is intended for large instances. In particular, the higher the number of clusters (p in the problem definition), the more effective is this method: with $p = 2$ for instance, this method is useless, as it degenerates in the normal VNS applied on the complete original problem.

Example Figure 4.8 shows an instance of the k-CmBCP that will be used to describe the steps that VNDS performs.



$n = 8$, $m = 6$, $p = 3$

Figure 4.8: An instance of the k-CmBCP.

Consider now Figure 4.9: again, right nodes are omitted, replaced by the cost of the solution. Let us suppose the solution at the left of the figure, with cost = 19, is reach by a local search method. We now apply the Decomposition Search. First, α is computed for each pair of clusters: in our case $a_{(1,2)}$ has the highest value, so c_1 and c_2 are elected to enter the subproblem. We now have only two clusters to work with; the number of left nodes is reduced to 6 and the cost of this solution (y) is 16. A local search method is applied to the subproblem, obtaining in few iterations a local minimum y' of cost 12. Finally the new solution y' is integrated in the original solution of the complete problem, obtaining a new, better, solution of cost 15.

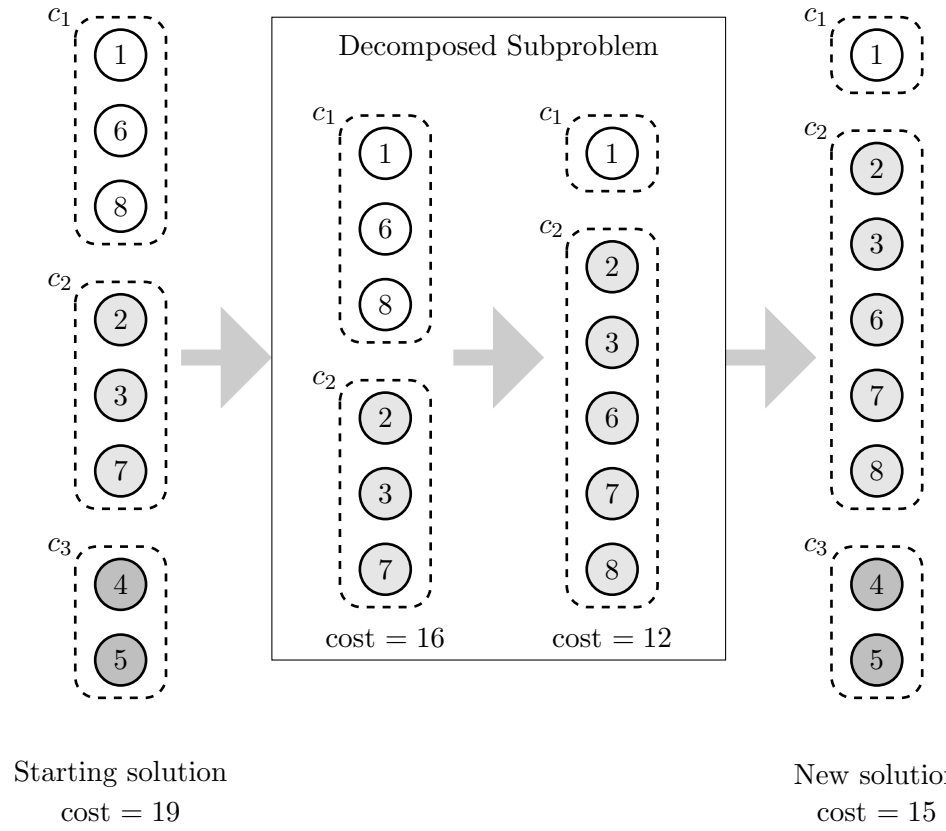


Figure 4.9: Steps of VNDS for solving the k-CmBCP.

While this was a toy example, it is easy to understand how this approach is very fast in very big instances too, especially if p is high. Furthermore, the particular structure of the subproblem makes it very easy to find global optima within itself. Thus we can obtain easily an exact optimization for a

part of the solution.

Unfeasible Search

One of the key words in search methods is *diversification*, that is the exploration of regions of the search space far away from local optima. VNS tackles it nicely, thanks to the `Shake` procedure, combined with the enlargement of the neighborhood. However we investigated another approach, that is not so common in literature, but seems promising to escape from local optima. Unlike the Decomposition Search, the Unfeasible Search is executed after the VNS has finished a complete cycle (k_{max} has been reached). It is similar to the *Variable Space Search* ([27]), in which the formulation of the problem is changed and the local search continues in the new generated space. It is a method that can be incorporated in every meta-heuristic, not necessarily in Variable Neighborhood methods. Its goal is simply to add diversification to the search. Its idea is to make the problem unfeasible and move inside the new unfeasible region for a given number of iterations. At the end it returns back to the feasible region, hopefully ending up in a solution far away from the previous one.

Algorithm 13 Steps of Unfeasible Search.

1: **procedure** UnfeasibleSearch(s, p, t_d)
2: $p \leftarrow p + 1$
3: $\sigma \leftarrow$ LocalSearch(s, t_d)
4: $s' \leftarrow$ BestMerge(σ)
5: **return** s'
6: $p \leftarrow p - 1$
7: **end procedure**

The goal of this method is very similar to the one achieved by the Random Restart, in which local search is applied iteratively from solutions generated at random. But Random Restart is much faster than this method. However our intentions go further than this goal. Other than experiment with new approaches, we tried to develop a method that, during its execution, could explore promising parts of the solution and, hopefully, bring them back to the feasible region. This idea will be clearer when we shall explain it in the next paragraphs.

The modification we apply to the problem is very simple: we allow to group left nodes in one more cluster. The problem changes, from an instance

$P = (I, J, p)$ to $P' = (I, J, p + 1)$. The solutions of P' are generally better than the ones of P (see Proposition 1).

In order to understand which space is represented by P' with respect to P, consider Figure 4.10.

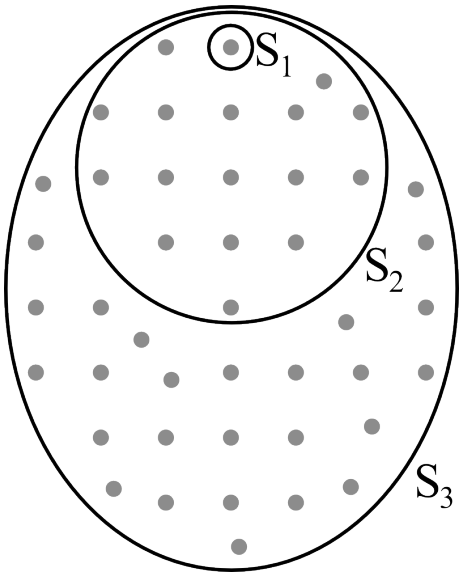


Figure 4.10: Representation of the solution spaces varying the number of cluters (p).

The bipartite graph $G = \langle I, J, E \rangle$ representing an instance of the problem is always the same. With S_k we indicate the solution space of the instance when k clusters are allowed. S_1 has only one solution: all the nodes together in the same cluster. If k increases, so the solution space does. Spaces are represented sequentially one inside the other to express the fact that a smaller k is a particular case of a bigger one, if we allow some clusters to be empty. Thus P is a particular case of P', in which one cluster is empty. This consideration is recursive for each k, so that at the end, S_1 is a particular case of every S_k with $k > 1$.

In Figure 4.11 we give a graphical representation of the behavior of the method. Given a local minimum s^* returned by the VNS on region S_p, we switch to the unfeasible region by allowing one more cluster. We move in the new region S_{p+1} with a Local Search procedure for a given number of iterations. Then we return to the feasible region S_p by combining two clusters together, obtaining solution s.

In the previous steps, the most delicate ones are the crossings of the border. The first move, from S_p to S_{p+1} is trivial. It is like s^* is already

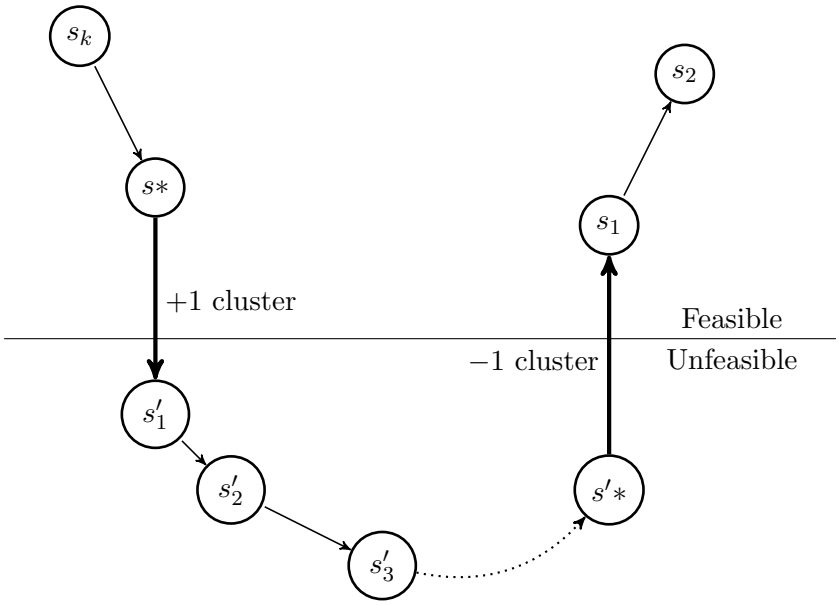


Figure 4.11: Representation of the Unfeasible Search method.

in S_{p+1}: it is only a non-optimal solution, since one cluster is empty. It is sufficient to let the descent method choose which nodes to put into the new cluster. The opposite move, on the contrary, has received more attention. As stated before, our idea was to be able to keep promising parts of the solution at the end of the unfeasible search. It could happen, in fact, that the search in S_{p+1} outlined few good clusters that should be kept intact. A union of two random clusters could destroy what we have built so far. Thus we decided to merge two clusters such that the added cost is minimized. This acts like a separated local search inside the `UnfeasibleSearch` method. In this particular local search, called `BestMerge`, we use a neighborhood equal to the whole solution space so that we are able to return the global optimum in one step. A move consists in merging two clusters. So the size of the neighborhood, that we indicate with N_{p+1}, is the number of all possible unordered pairs that can be extracted from a set of $p+1$ elements:

$$|N_{p+1}| = \binom{p+1}{2} = \frac{p(p+1)}{2}$$

Algorithm 14 shows the steps of this local search: c_i indicates the cost of cluster i, that is the number of additional nodes required to make it a biclique. This move is quite slow, but since it is executed only once, it does not affect the total time so much.

Algorithm 14 Steps of `BestMerge` procedure.

1: **procedure** BestMerge(σ)
2: $P \leftarrow$ all possible unordered pairs from σ
3: **for** $\{i, j\} \in P$ **do**
4: $c_{\{i,j\}} \leftarrow$ cost of merged pair $\{i, j\}$
5: **end for**
6: $p' \leftarrow arg \min_{\{i,j\})\in P} \{c_{\{i,j\}} - (c_i + c_j)\}$
7: `merge`(i, j)
8: **end procedure**

4.3 Column Generation

When a problem formulation has many variables and a reduced number of constraints, *Column Generation* ([28]) may be beneficial. Column Generation is a procedure used for tightening an LP relaxation, in a complementary way with respect to *Cutting Planes*.

The problem must be formulated in this form, called *Master Problem*:

$$z_{MP} := \min \left\{ \sum_{j \in J} \gamma_j \lambda_j : \sum_{j \in J} \mathbf{a}_j \lambda_j \geq \mathbf{b}, \lambda_j \geq 0 \, \text{integer}, j \in J \right\}$$

where J contains a large number of points characterizing the feasible region and each λ_j contains the corresponding variables. We then consider the LP relaxation of the Master Problem, where λ_j is allowed to assume any value ≥ 0.

Briefly, the method works this way. The majority of the columns are left out of the LP relaxation because there are too many columns to handle efficiently and most of them will have their associated variable equal to zero in an optimal solution anyway. Thus a subset of columns is selected, at least enough to make the problem feasible: we obtain the *Restricted Master Problem*. We solve its relaxation, obtaining an optimal solution for the restricted LP version of the Master. Then, to check the optimality of the solution, a subproblem, called the *Pricing Problem*, which is a separation problem for the dual LP, is solved to try to identify columns to enter the basis. Given a set of dual values, either identify a column that has a favorable reduced cost, or indicate that no such column exists. If such columns are found, they are added to the RMP and the LP is re-optimized.

The formulation of the k-CmBCP has already been written in Section

3.4, called (model-4). It is repeated here, followed by further details. Let us recall that t represent a cluster, and c_t its cost. λ_t is the variable that indicates whether we take or not a cluster in our solution. Then, the Master Problem is as follows.

Master Problem (MP)

$$\min \quad \sum_{t \in T} c_t \lambda_t \tag{4.1}$$

$$\text{s.t.} \quad \sum_{t \in T} a_{it} \lambda_t = 1 \qquad \forall\, i \in I \tag{4.2}$$

$$\sum_{t \in T} \lambda_t = p \tag{4.3}$$

$$\lambda_t \in \{0, 1\} \qquad \forall\, t \in T \tag{4.4}$$

This problem can be relaxed into a set covering problem, since from a covering solution is always possible to construct a partitioning. Relaxing the integrality constraint, we get:

$$\min \quad \sum_{t \in T} c_t \lambda_t \tag{4.5}$$

$$\text{s.t.} \quad \sum_{t \in T} a_{it} \lambda_t > 1 \qquad \forall\, i \in I \tag{4.6}$$

$$\sum_{t \in T} \lambda_t = p \tag{4.7}$$

$$\lambda_t \in [0, 1] \qquad \forall\, t \in T \tag{4.8}$$

Let π be the vector of dual multipliers of constraint (4.6), and let μ be the dual multiplier of constraint (4.7). The Pricing Problem is as follows.

Pricing Problem (PP)

$$\min \quad \sum_{(i,j) \in \bar{E}} z_{ij} - \sum_{i \in I} \pi_i x_i - \mu \tag{4.9}$$

$$\text{s.t.} \quad z_{ij} \geq x_i + y_j - 1 \qquad \forall \{i, j\} \in \bar{E} \tag{4.10}$$

$$x_i \leq y_j \qquad \forall \{i, j\} \in E \tag{4.11}$$

$$x_i, y_j, z_{ij} \in \{0, 1\} \qquad \forall i \in I,\, \forall j \in J \tag{4.12}$$

An alternative formulation for the pricing problem is as follows:

$$\min \quad \sum_{(i,j)\in\bar{E}} z_{ij} - \sum_{i\in I} \pi_i x_i - \nu \tag{4.13}$$

$$\begin{aligned} \text{s.t.} \quad & z_{ij} \geq x_i + x_l - 1 && \forall\{i,j\} \in \bar{E}, \forall\{l,j\} \in E \\ & x_i, z_{ij} \in \{0,1\} && \forall i \in I, \forall j \in J \end{aligned} \tag{4.14}$$

This formulation has m variable less (the y_j variables), but it has more constraints (4.14).

The interpretation of (PP) is as follows. The column we are trying to add represents a cluster t, defined by a subset I. This set induces a subgraph G_t of $\langle I, J\rangle$. Three sets of variables are present:

$$x_i = \begin{cases} 1 \text{ if session } i \text{ belongs to the cluster} \\ 0 \text{ otherwise} \end{cases}$$

$$y_j = \begin{cases} 1 \text{ if client } j \text{ belongs to the subgraph } G_t \\ 0 \text{ otherwise} \end{cases}$$

$$z_{ij} = \begin{cases} 1 \text{ edge } \{i,j\} \text{ is required to make } G_t \text{ a biclique} \\ 0 \text{ otherwise} \end{cases}$$

Constraints 4.10 and 4.11 make the variables consistent. The first one forces z_{ij} to assume value 1 when i and j are both in G, but no edge connects them (we pay the additional edge). The second one forces y_j to be 1 when at least one edge connects j to a node i belonging to t. Note that x represents the characteristic vector of t. The objective function tries to minimize the cost of cluster t, after having subtracted from it a weighted sum of the nodes belonging to t. The weights are the dual multipliers obtained from MP. The problem is similar to the *Maximal Quasi-Biclique* defined in [29], where it has been applied to protein-protein interactions.

Solving the PP to optimality is time-consuming, considering that it must solved multiple times, one for each generated column. Computational issues arising in Column Generation represent a substantial part in Discrete Optimization research ([30]). Thus we have designed a heuristic to speed up the lengthy procedure.

Algorithm 15 shows the step of the method. With r_λ we indicate the reduced cost of the column identified by λ. Once the (relaxation of the) RMP is solved, the procedure `heuristicPricing` is called. If the column returned has non-negative reduced cost, then the `exactPricing` is called. This step

Algorithm 15 Column Generation procedure.

```
1: procedure ColumnGeneration
2:     repeat
3:         π, μ ← solveRMP
4:         λ ← heuristicPricing(π, μ)
5:         if r_λ ≥ 0 then
6:             λ ← exactPricing(π, μ)
7:         end if
8:         if r_λ ≥ 0 then
9:             addColumn(λ)
10:        end if
11:    until r_λ ≥ 0
12: end procedure
```

is necessary, since the heuristic method does not guarantee the absence of a column with a negative reduced cost and halting the procedure before all the columns are generated yields a problem which is not, in general, a relaxation of the original one. The following step checks whether the column has negative reduced cost again. If condition is true, the column is added to the RMP. The entire loop is repeated until no eligible column was found by any of the two methods.

We now discuss the heuristic for solving the PP. We use λ to indicate a solution in the search space and r_λ its reduced cost. In the local search model, a solution is a cluster and it is defined by the set of nodes that form the cluster. In mathematical terms, it can be expressed with a set of binary variables x_i, with $i \in I$, which have value 1 if node i is in the cluster, in the same way as in the PP formulation.

Algorithm 16 shows the steps of the method. The basic structure is a VNS, but some refinements are needed to make it suitable for the problem. First, the starting solution must be chosen. Random generation or an empty set are both reasonable choices. However we have decided to choose the starting solution based on the cluster that was generated in the previous call. We consider the last generated cluster and selects all and only the nodes that have not been taken. The reason for this expedient is to avoid generating similar columns at the first iterations. A local search method, being non-optimal, could in fact move in valleys similar to the one of the previous local optimum. Our method starts each time from the farthest

Algorithm 16 Heuristic to solve the Pricing Problem.

```
1: function heuristicPricing(π, μ)
2:     k ← 1
3:     λ ← startingSolution
4:     repeat
5:         λ' ← Shake(λ, k)
6:         λ'' ← BestImprovement(λ')
7:         if r_λ'' < 0 then
8:             return λ
9:         else
10:            NeighborhoodChange(λ, λ'', k)
11:        end if
12:    until k > k_max
13:    return λ
14: end function
```

solution from the one returned in the last call.

The other methods of `heuristicPricing` have already been discussed in the previous section and need no explanation here. The only addition to a basic VNS is the `if` at line 7: if, after a complete descent, the reduced cost is already negative, the procedure terminates, returning the solution found. This trade-off between quality of solution and time has proved to be critical in the overall performance of the CG procedure. Letting k increase beyond 1 generally leads to solutions of slightly better value, if any. However, the time required is, in average, k_{max} times greater. While this is an acceptable condition for a heuristic when finding an upper bound for a problem, in Column Generation, considered as a whole procedure, the performance is worsen considerably.

Given the previous consideration, the choice for the `bestImprovement` procedure as descent method is dictated by the need of finding the best solution possible before reaching line 7 and by the limited size of the neighborhood ($|N(\lambda)| = n$). The risk of a local search is to generate too many columns, by selecting solutions far from the optimal -but with negative reduced cost- each time. `bestImprovement` reduces the explosion of columns, especially in the first iterations, when the dual multipliers have high and very different values.

4.4 Branch-and-Price

Once we have two methods to obtain, respectively, an upper and a lower bound for an instance of the problem, a *Branch-and-Bound* ([31]) approach would guarantee to find the integer optimal solution. Since the relaxation method we have chosen is Column Generation, the branching algorithm is called *Branch-and-Price*, which presents fundamental differences with a standard branching approach (see [32]).

In fact an LP relaxation solved by column generation is not necessarily integral and applying a standard branch and bound procedure to the restricted master problem with its existing columns will not guarantee an optimal (or feasible) solution. After branching, it may be the case that there exists a column that would price out favorably, but is not present in the master problem. Therefore, to find an optimal solution we must generate columns after branching.

In developing a B&P method, many computational issues arise, both for the LS heuristic and the CG. Some of them will be presented in Chapter 5, while here we discuss the most important conceptual aspect: how to do branching. The typical problem found in B&P is that branching on the variables of the Master Problem (called λ from now on) requires to add constraints to the formulation of the MP. This fact induce a different PP to be solved at each node, that cannot be derived automatically.

We outlined three possible variables on which branching could be done:

1. Integer variables x_i, whose value is the number of the cluster to which node i belongs.
2. Binary variables g_{il}, expressing whether or not node i and j must be grouped in the same cluster.
3. CG variables λ, representing a cluster.

We have implemented the second choice and, without going into details, we briefly discuss the other two possibilities below. To avoid generating confusion with the bipartite graph representing an instance of the k-CmBCP, a node of the B&P tree is called child, except when it is the first one, called root.

Integer variables x_i induce many symmetrical nodes to be generated in the B&P tree. A symmetry breaking method should be implemented. Furthermore, their meaning is not directly translatable into the CG formulation: stating that node $\hat{i}$ must be in cluster $\hat{k}$ does not impose any condition on

how the clusters should be generated.

Variables λ present a problem whenever a child is generated by imposing $\hat{\lambda}$ to assume value 0. In the PP there is no way of expressing this constraint by means of linear constraints. Besides, constraints on λ disrupt the neighborhood structure of a local search algorithm by isolating some solutions: moving from a solution can become impossible.

We now describe how branching on binary variables g_{il} has been designed. The children generated at each node of the B&P tree are two: one for $g_{il} = 0$ (left branch) and the other one for $g_{il} = 1$ (right branch). Each of them requires different processing. The maximum depth of the B&P tree is:

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

Forbidding two nodes i and l to be grouped together is achieved without many efforts. In the LS heuristic moving i into another cluster is forbidden if the destination cluster contains l. Note that, when branching rules becomes numerous as the depth of the tree increases, some solutions in the neighborhood will be unreachable in one simple move and the upper bound found by the heuristic can have higher value. A workaround to this problem is obtained by evaluating more moves at once and considering the branching rules at the end only. This, however, would require a lot of computation time. We have decided to maintain the first method, because it rarely happens that so many nodes are explored in the B&P tree without reaching termination. Besides, if such a situation arises, the VNS will terminate faster.

In order to forbid i and l to be grouped together in the CG procedure, it is sufficient to add a linear constraint to the PP:

$$x_i + x_l \leq 1 \quad i, l \in I$$

The accordance with the branching rule is maintained if we start from a RMP with no columns containing both i and l, as generated columns will never brake the rule thanks to the added constraints. To do that, at the beginning of the CG, we populate the columns of the RMP by extracting them from the solution found by the LS heuristic, that is known to be correct with respect to the rules.

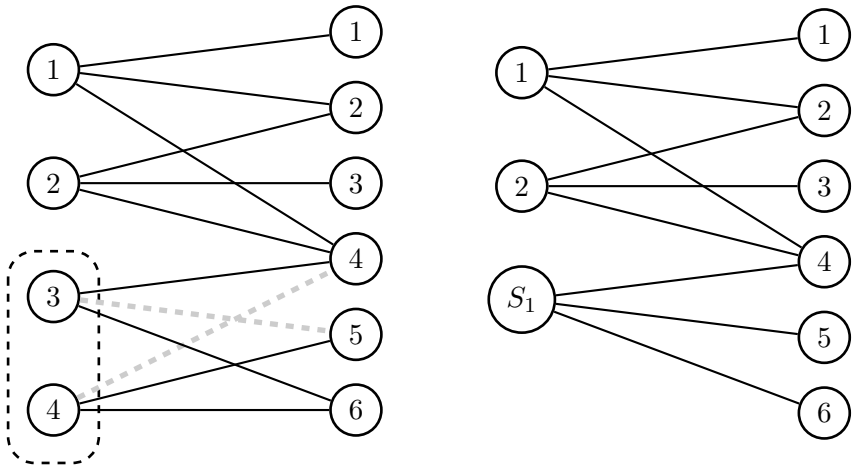
On the other hand, forcing i and l to be grouped in the same cluster has been realized with a different method. No constraints are added to the models, but the instance is changed. Once the instance is changed in a node, it

is passed to the children and will propagate to the descendants, with further transformations. The method is based on the following consideration:

- if i and l must be grouped together, in a feasible solution every cluster eithers contain both of them or none of them. Besides, the solution always pays the additional edges needed to complete the biclique induced by i and l.

We introduce now the concept of *super-node* as a node representing a set of nodes. The transformation of an instance with left shore of size n and right shore of size m is the obtained with following steps:

1. Nodes i and l are replaced by a super node S_1: n is decremented by 1 and all the nodes $\in I$ are reordered accordingly.
2. S_1 is connected to all the nodes contained in the biclique induced by i and l, that is all the nodes $\{j \in J | \{i, j\} \in E \text{ or } \{l, j\} \in E\}$.
3. The number of additional edges required to make the subgraph induced by i and l a biclique in the original graph constitute the cost of the new instance.



Original instance cost: 0

(a)

New instance cost: 2

(b)

Figure 4.12: Transformation of an instance, using the concept of *super-node*.

Figure 4.12 shows an example of what we have just described: nodes 3

and 4 are combined in a super-node S_1; the cost of the new instance is 2, equal to the cost of the cluster induced by 3 and 4 in the original graph; S_1 is connected to all the edges of the original biclique.

The cost of an instance is an intrinsic attribute that must be considered throughout the whole tree when computing the bounds: it is added to the cost of any solution on a given instance. The previous procedure happens at each child generated by a branching rule that forces two nodes together. Hence it is recursive: super-nodes grow bigger as the depth of the tree increases and can contain many nodes of the original graph. The cost of a transformed instance is added to the cost of the original one. At each child, super-nodes are considered as normal nodes and combined with the same steps listed before: only a mapping is needed to be able to convert a solution on a given instance back to the original problem of the root node.

The only modification required to the solution methods is the following: when computing the cost of a cluster, the cardinality of the super-node must be considered. When an edge $\{i, j\}$ is paid, the cost is not 1 anymore, but it must be multiplied for the cardinality of i. Using the binary variables $z_{i,j}$ to indicate whether an edge is paid or not and indicating with d_i the cardinality (dimension) of super-node i, the objective function of the model of the problem becomes:

$$\sum_{\{i,j\}\in\bar{E}} d_i \, z_{i,j}$$

The size of the left shore is decremented by 1 every time a pair of nodes is required to be grouped together: this feature allows to close a child if the number of nodes of the incumbent instance is reduced to p after $(n-p)$ right branches have been generated.

Note that the procedure described above transforms the k-CmBCP into the generalized version with weights on nodes, mentioned in Section 2.3. Thus all the methods that we have implemented are designed for the generalized version too.

The two methods described above for the branching rules have different effects on the solution of a node: transforming an instance through a super-node makes it smaller, thus easier to solve; adding a linear constraint makes local search faster, by reducing the neighborhoods, but yields an IP model harder to solve. As the tests will show in Chapter 6, our B&P approach requires a computation time at each node that tends to shorten as the depth of the tree increases.

The last aspect we discuss is the criterion used to select the pair of nodes $\{i, j\}$. Conforming to a traditional approach, the variable λ that presents the most fractional value at the end of the CG is considered. Since λ can contain more than one node, two nodes are extracted. If λ represents a cluster formed of one node only, another node is chosen randomly, taking into consideration the branching rules active at the current child.

Chapter 5

Implementation

The whole system has been implemented in COMET. In this chapter we introduce the language and describe its features. The first part is suggested to the reader that wants to acquire full understanding of the following sections. Next, our implementation is explained. First we give a general overview of the program, then, for each part, the relative code is shown and discussed in detail.

5.1 The Comet language

COMET ([33]) is an hybrid optimization system, combining constraint programming, local search, and linear and integer programming. It is also a full object-oriented, garbage collected programming language, featuring some advanced control structures for search and parallel programming. It has been developed by Pascal Van Hentenryck, Professor in the Computer Science Department and director of the optimization lab at Brown University, and Laurent Michel, Assistant Professor in the Computer Science and Engineering Department at the University of Connecticut.

We have decided to use COMET as the main language for the implementation of the algorithms for several reasons. The main one is that it drastically simplifies the design and implementation of local search algorithms. While this is a very useful feature, it is just one of the many that COMET provides. It allows very efficient implementations. It also respects the freedom of the programmer: while it provides many constructs that avoid to write boring code, the implementation of an algorithm is not guided or restricted in any way. It is completely up to the programmer to design it in every detail. In this way COMET can be seen as a enhanced C++ language, meaning that

it is empowered with built-in structures that save time to the programmer.

One of the main innovations of Comet is Constraint-based Local Search, a computational paradigm based on the idea of specifying local search algorithms as two components: a high-level model described in terms of variables, constraints and objective functions; a search procedure expressed in terms of the model at a high abstraction level. Constraint-based local search makes it possible to build local search algorithms compositionally, to separate modeling from search, to promote re-usability across many applications, and to exploit problem structure to achieve high performance.

Comet is quite new at the time of this work and it is constantly under development. The draw-back of using such a language is that it changes pretty quickly and the updated documentation is often published uncompleted or late with respect to a new system release. Besides, some features are yet to be implemented and the developer may be compelled to make unwanted implementation choices.

This chapter wants to introduce the language to the reader, to outline its remarkable features and to provide a sort of tutorial through a few examples. An exhaustive guide is out of our scope, so we limit our introduction to the key concepts that have been used for this work (local search and mathematical programming). In the document we refer to the version 2.0-rev0 of the language, released in September 2009.

5.1.1 The basics of the language

The language is similar to C++ and Java and it shares some features with both of them.

Comet has three primitive types: `int`, `float` and `bool`. These primitive types are given by value in function/method parameters. The object versions of these, which are given by reference, are: `Integer`, `Float`, `Boolean`.

Strings are implemented by the `string` immutable object. Many methods that work on strings are provided to: concatenate, extract substrings, finding and/or replacing parts, trimming, converting to int and float.

Arrays behave like in C++: multi-dimension arrays and matrices can be declared too. The main difference is that a `range` is always associated to an array to specify the valid indexes allowing to retrieve or modify the entries. A range corresponds the an integer interval $[a..b]$ between two integers a and b. A range is an immutable object.

The usual data structures (dictionaries, sets, stacks and queues) are

available in COMET.

sets A collection of elements of type `T`. Basic operations are: inserting/removing an element, checking whether an element is present, union/intersection/difference of sets.

dictionaries A dictionary (keyword `dict`) is a set of entry $< key, value >$ and the value can be retrieved with the key. The keys and values can be of any type but must be specified at the creation of the dictionary.

stacks Stacks are a let-polymorphic data type. It is used to add elements to the end (with the `push` method). The elements can be removed from both ends of the stack (`pop()` and `popBottom()`) and can be accessed directly at any position `i` with `i` operand.

queues Queue is also a let-polymorphic data type. Contrary to a `stack`, a `queue` can grow from both ends with the `enqueueBack` and `enqueueFront` methods. Also, contrary to a `stack`, elements cannot be accessed by their position.

Loops ad flow control are the same as C++ and Java: `if`, `for`, `while`, `do while`. Particularly useful is the `forall`, that allows to iterate over the values of a `range` or a `set`; it is also possible to make a condition on the iteration.

COMET has several types of selectors, that allow to select randomly or according to a probability or an evaluation function an element in a set or a range:

- `select` is the most basic one, it selects randomly.
- `selectMin/selectMax` chooses an element according to an evaluation function. Some randomness can also be introduced by selecting randomly between the k bests.
- `selectPr` chooses an element according to a probability density function.
- `selectFirst` is deterministic and selects the smallest lexicographic element satisfying a set of conditions.
- `selectCircular` is a deterministic selector with a state such that several executions of it consider elements in a circular way.

Functions behave as in C++. Arguments of functions are given by reference for every objects and are given by value for the basic types `int`, `float`

and `bool`.

Classes follows closely the syntax of a Java class. Some characteristics of Comet classes are:

- No modifier on the class or on the methods (everything is public).
- Support methods or constructors overloading.
- No default constructor like in Java.
- All the instance variable can only be accessed from inside the class (getter and setter methods must be implemented to modify it or communicate from outside).
- The instance variables cannot be initialized from outside of the body of a constructor or a method.
- `this` object is available inside the implementation of a class but it cannot be used to access a instance variable like `this.intanceVar`.
- Like in Java there are an interface mechanism, inheritance mechanism and pretty print of Objects.

There is an exception mechanism similar to the one of Java. Every Comet object can be thrown as an exception and the exception throw can be surrounded with the classical `try/catch`.

Finally full support for threads is available.

5.1.2 Constraint Programming

Constraint Programming (CP) is a programming paradigm where relations between variables are stated in the form of constraints. It is used to solve effectively large, particularly combinatorial, problems especially in areas of planning and scheduling. It is a form of *declarative* programming, meaning that the logic of a computation is expressed without defining its control flow.

Each variable is assigned a known domain. Then, constraints are added to the problem, restricting the domains. The goal is to find an assignment of values to the variables that do not violate any of the constraints. The search of a solution can be completely non-deterministic, or guided. The separation between the *model* of the problem and the *search* method is a peculiar feature of CP.

A CP model in Comet has the structure shown in Listing 5.1. One can see that there is a clear separation between the modeling and the search part. To enumerate all the solutions, the `solve` keyword must be replaced

by `solveall`.

```
1 import cotfd;
2 Solver<CP> cp();
3 // declare the variables
4 solve<cp>{
5   // post the constraints
6 } using{
7   // non deterministic search
8 }
```

Listing 5.1: Structure of a CP program.

Variables Discrete integer variables are the most commonly used. They must be given a CP solver and a domain. The domain of a variable specifies the set of values that can be assigned to the variable. Next example declares a variable `x` with the integer interval domain [1..10].

```
var<CP>{int} x(cp,1..8);
```

Besides discrete integer variables, COMET also allows to declare boolean and float variables:

```
var<CP>{bool} b(cp);
var<CP>{float} f(cp,1,5);
```

The domain of the boolean variable `b` is $\{\text{true}, \text{false}\}$ and the domain of the rational variable `f` is the rational interval $[1, 5]$.

Constraints The constraints act on the domain store (current domain of every variables) to remove inconsistent values. Behind each constraint, there is a sophisticated filtering algorithm that prunes the search space by removing values that don't participate in any solution of the constraint. As soon as a domain becomes empty, the domain store fails, which means that there is no possible solution and the search must backtrack to a previous state and try another decision. To summarize, a constraint must implement two main functions:

- *Consistency check*: verify that there is a solution to the constraint, otherwise the constraint can tell to the solver that it must backtrack.
- *Filtering of the domains*: remove inconsistent values.

A constraint can be posted with the `post` method on the CP solver.

Comet allows you to post constraints implying complex arithmetic and logical expressions like in the next example. Note that (`x[i]==4`) is converted to a 0/1 variable and that `=>` posts a logical implication constraint that must be true.

```
cp.post(sum(i) (x[i]==4)*dim[i] == totalDim);
cp.post(y>z => a<b);
```

CP Search The search in constraint programming consists in an non-deterministic exploration of a tree with a backtracking search algorithm. The default exploration algorithm is a depth first search. Next example explores with a depth first all the combinations of three 0/1 variables.

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..3](cp,0..1);
solveall<cp>{
}using{
   label(x);
   cout << x << endl;
}
```

It produces the output

```
x[0,0,0]
x[0,0,1]
x[0,1,0]
x[0,1,1]
x[1,0,0]
x[1,0,1]
x[1,1,0]
x[1,1,1]
```

The search in Comet can be deterministic, which means that at each node of the search tree there is the possibility to control the sub-trees under this node. The next variable to instantiate can also be decided on the fly for each choice point. Next example uses a randomized selector `select` to choose the index of the next variable to instantiate.

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..3](cp,0..1);
```

```
UniformDistribution distr(0..1);
solveall<cp>{
}using{
  while(!bound(x)){
    select(i in x.rng():!x[i].bound()){
      int v = distr.get();
      try<cp> cp.label(x[i],v); | cp.diff(x[i],v);
    }
  }
}
```

Optimization in CP Constraint Programming can solve optimization problem. To solve an optimization problem, each time a solution is found during the search, a constraint stating that the next solution found must be better is dynamically added. By construction, the last solution found is optimal. The structure of a COMET program minimizing an expression is shown in Listing 5.2.

```
1  import cotfd;
2  Solver<CP> cp();
3  //declare the variables
4  minimize<cp>
5    //expression or variable
6  subject to{
7    //post the constraints
8  } using{
9    //non deterministic search
10 }
```

Listing 5.2: Minimization in CP.

5.1.3 Constraint Based Local Search

In this section we present the COMET Constraint Based Local Search (CBLS) component and how it can be used to develop local search solutions to different problems.

Variables

A key idea of Comet's local search module is the use of differentiable objects. Objects of this kind can be queried, in order to find out about the effect of a value assignment on the object's evaluation.

Incremental variables are defined in conjunction with a local solver and they are the building blocks to form all kinds of differentiable objects (expressions, invariants, constraints and objective functions) within the local solver. Typically, the local solver provides the mechanism for managing dependencies between variables that are constituents of the same differential object, and for updating the evaluation of differential objects after changes in the values of the incremental variables.

Discrete integer variables are the most commonly used. `set`s, `range`s or `enum`s can all be used as domains. There are three types of incremental variables: integer, floating point and set over integers or floating point. Some standard ways to define incremental variables are contained in Listing 5.3. Note that assignment to incremental variables is done using the `:=`

```
1 import cotls;
2 Solver<LS> m();
3 range Domain = 1..n;
4 var{int}       a(m,Domain);
5 var{float}     f(m) := 3.14;
6 var{set{int}}  s(m) := {2,3,5,7,11};
```

Listing 5.3: Definition of incremental variables.

operator. Another useful operator is the `:=:` which can be used to swap two incremental variables.

Invariants

Invariants are one-way constraints of declarative nature: they specify what to ensure, without specifying how. They are expressed in terms of incremental variables and expressions, and specify a relation which must be maintained under assignments of new values to the participating variables.

More specifically, an invariant is an instruction of the form:

```
y <- exp
```

where `y` is an incremental variable and `exp` is an expression. COMET guarantees that, at any time during the computation, the value of variable `y` is equal to the value of the expression `exp`. For example, the code

```
var{int} x(m);
var{int} y(m) <- x^2;
```

ensures that, at all times, variable `y` will equal the square of variable `x`. An important thing to point out is that invariant declarations are not allowed to form cycles. Invariants can be formed using incremental variables, constants and standard operators supported by COMET: unary (`abs`, `floor`, `ceil`, `-`), binary `(+, -, *, /, %, ^, min, max, &&, ||)`, and relational `(>, <, <=, >=, ==, !=)`. One can also use the aggregate counterparts of binary operators, defined over sets or ranges: `sum(k in S)`, `prod(k in S)`, `min(k in S)`, `max(k in S)`, `and(k in S)`, `or(k in S)`.

We show some of the available uses in the following examples.

```
var{int} m[i in R] <- f[g[i]]
```

indexes the array `f` of incremental variables with the incremental variable `g[i]`. Semantically the meaning of this statement is straightforward, although things are non-trivial behind the scenes: array `m` has to be updated when either `g[i]` changes or `f[v]` changes.

The statement

```
var{int}     a[R];
var{int}[] f = count(a);
```

is equivalent to:

```
var{int} a[R];
var{int} f[d in D] <- sum(i in R) (a[i] == d);
```

However, the implementation of `count` is much more time- and space-efficient.

Constraints

Constraints in COMET are built on top of invariants, and are central in the architecture of the language.

Every constraint implements methods that return information about the satisfiability of the constraint. The total number of violations is given by the `violations()` method. Function `decrease(x)` has the same semantics as the `violations(x)` method, and returns the maximum decrease in the

constraint's violations by assigning the incremental variable `x` to any other value in its domain.

Two often used methods are:

- `getAssignDelta(x,v)` returns the difference in violations by assigning variable x to value v;
- `getSwapDelta(x,y)` returns the difference in violations by swapping incremental variables x and y.

Finally, the method `post` is used for adding a constraint to a constraint system.

Constraints are characterized by their number of violations, which is a measure of how far from satisfaction the constraint is. There are three common ways to assign violations to a constraints:

Decomposition-Based Violations sum of violations of the individual constraints that comprise the constraint system.

Variable-Based Violations minimum numbers of variables that need to change, in order for the constraint to be satisfied.

Value-Based Violations in constraints that specify lower or upper bounds on the number of occurrences of particular values in the variable assignment, one can base the number of violations on how well these bounds hold for different values in the domain.

Search

Once the model of the problem has been defined, it is up to the algorithm designer to implement a search procedure. Generally the search is guided by the constraints or by an objective function. The programmer is free to experiment with several variations, since the model remains the same.

A typical search procedure is shown in Listing 5.4.

The previous example moves in the search space until a stop criterion is met (a limit of iterations in this case). Every move evaluates the change in the ConstraintSystem `S` violations when swapping the values of the variables between any pair of nodes.

We now introduce a local search structure that greatly facilitate the development of local search algorithms: *Solutions*.

```
1 while (iteration < limit) {
2   selectMin (i in Nodes, j in Nodes,
3     delta = S.getSwapDelta(x[i], x[j]): i < j) (delta) {
4     x[i] :=: x[j];
5   }
6   iteration++;
7 }
```

Listing 5.4: A typical search procedure.

Solutions

Given a local solver, *Solutions* can be used to store the current assignment of all incremental values not defined through invariants. Solutions are first-class objects, which means that they can be passed as parameters, stored in different data structures, or returned as results of function calls.

Typically, one can define a `Solution` object with a call

```
Solution s(m);
```

At any point of the execution of an algorithm one can restore a solution `s` using the `restore` method.

```
s.restore();
```

This will restore all variables to their state at the point when solution `s` was created. The solver will then automatically update any other object that depends on the incremental variables of the solution. In addition to restoring a whole solution `s`, one can also query it to determine the values of specific variables. For instance,

```
x.getSnaphot(s);
```

returns the value of incremental variable `x` in solution `s`.

Solutions can greatly facilitate the coding of meta-heuristics that re-use already found solutions. For example, when using an intensification component to the tabu search, a solution is stored every time its value is better than the previous best solution and it is restored when the search procedure terminates.

5.1.4 Linear Programming

Comet provides a library to define linear programming models and to use a linear and mixed-integer solver. The library is named `cotln`.

Defining a model is straightforward (see Listing 5.5).

```
1 import cotln;
2 Solver<LP> lp();
3   // Define variables
4 minimize<lp>
5   // Define objective function
6 subject to
7   // Post all the constraints
```

Listing 5.5: Definition of an LP model.

The most useful methods of a `Solver` class `lp` are:

- `lp.getObjective().getValue()` retries the value of the objective function in the current optimum.
- `lp.updateLowerBound(var x, value)` updates the lower bound of a variable to the value *value*.
- `lp.updateUpperBound(var x, value)` the opposite of the previous one.

Support for *Mixed-Integer Problem* solvers has been added recently. It is sufficient to use `<MIP>` as the type of solver in the declaration of the model.

5.2 Implementation of the solution methods

In this section we present the program that we have developed to solve the k-CmBCP. We separate the code according to the three main classes: one for the Local Search, one for the Column Generation and one for the Branch-and-Price. This separation becomes natural even from a theoretical point of view: the Local Search and the Column Generation can be used independently from each other.

An overview of the whole framework as a simplified class diagram is shown in Figure 5.1. Only the main attributes and methods have been inserted in the diagram.

Our intention here is to expose how the main conceptual aspects have been translated into a program. Thus we limit our exposure to the key portions of the code, omitting the majority of the parts that make the final program run correctly. Furthermore, some lines will be skipped when not really needed, meaning that the listings written in this document cannot be considered correct programs.

5.2.1 Local Search model

A class is used to keep a model of the problem and its methods are called to perform every aspect of the search. Its declaration is shown in Listing 5.6 and 5.7.

Lines 3-9 declare the variables needed to store the instance of the problem. Names follow the conventions of previous chapters, in particular they are taken Multicast Partition problem:

- `n`: size of left shore (sessions);
- `m`: size of right shore (clients);
- `p`: number of clusters;
- `adj`: adjacency matrix, storing the edges;

Lines 11-13 declare the structures used for the local search. The only incremental variable engaged in the search is `x`, an array of `int`: each element corresponds to a session and its value is the number of the cluster in which it is grouped. The search is guided by `obj`, a function defined on `x` that computes the total cost of the solution. Note that this formulation is brief and it does not require any constraint. The drawback is that it contains many symmetrical solutions. However, the straightforward structure,

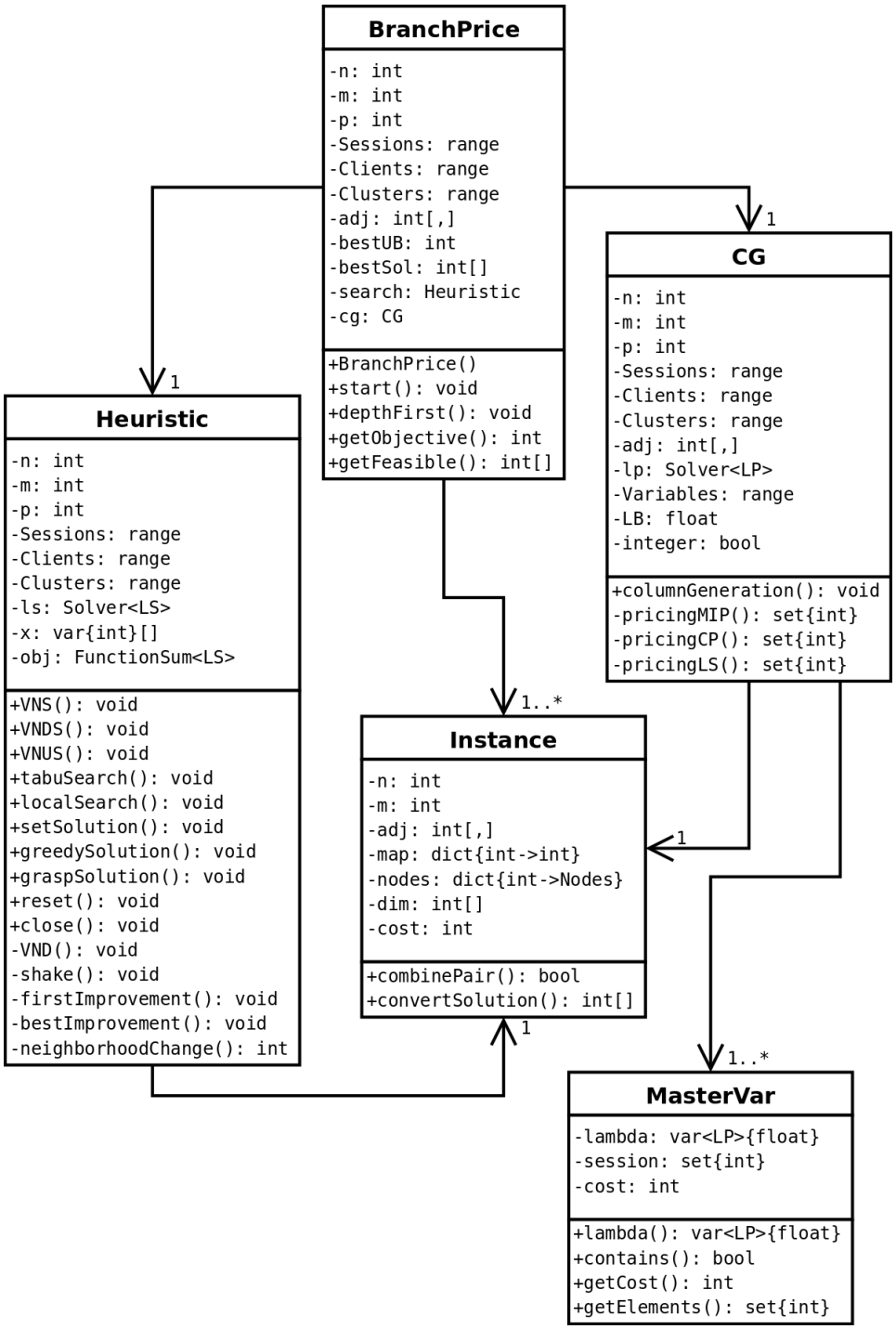


Figure 5.1: Class Diagram of the whole framework.

```
1  class Heuristic {
2
3    int n;
4    int m;
5    int p;
6    range Sessions;
7    range Clients;
8    range Clusters;
9    int[,] adj;
10
11   Solver<LS> ls;
12   var{int}[] x;
13   FunctionSum<LS> obj;
14
15   UniformDistribution cDistr;
16   bool[,] tabu;
17   Counter it;
18   Solution globalBest;
19   int globalBestCost;
20   int kMax;
21   int kMaxDescent;
22   int tabuLength;
23   stack{int[]} visited;
```

Listing 5.6: `Heuristic` class: Model of the problem.

```
24
25   Heuristic(int _n, int _m, int _p, int[,] _adj, bool _debug);
26
27   void VND();
28   void VNS(int mode);
29   void VNDS();
30   void VNUS();
31   void tabuSearch(int iterations);
32   void localSearch();
33
34   void setSolution(int[] sol);
35   void greedySolution();
36   void graspSolution();
37   void printSolution();
38   int[] getSolutionArray();
39   int getObjective();
40   int getIterations();
41   int getElapsedTime();
42   void reset();
43   void close();
44
45   void shake(int k, set{int} Variables, set{int} Domain);
46   void firstImprovement(set{int} Variables, set{int} Domain);
47   void bestImprovement(set{int} Variables, set{int} Domain);
48   int neighborhoodChange(Solution s, int cost, int k);
49   void findBest(int k);
50   void decompositionSearch();
51   void unfeasibleSearch(set{int} Variables, set{int} Domain);
52   void update();
53   void resetTabu();
54
55 }
```

Listing 5.7: `Heuristic` class: Methods.

focused on sessions, makes it easy to deploy it in a search procedure.

Finally lines 15-23 declare some utility variables to support the search and some parameters to tune it. The `cDistr` distribution is used for generating a starting solution: it generates a random number among the possible clusters (the domain of `x`). Next we find:

`tabu`:	the tabu list
`it`:	an iteration counter
`globalBest`:	the best solution ever found
`globalBestCost`:	the cost of `globalBest`
`kMax`:	max size of the neighborhood
`kMaxDescent`:	max size of neighborhood in the descent phase
`tabuLength`:	duration of a tabu assignment
`visited`:	a list of already visited solutions

The first method is the constructor of the class. It requires the instance data and a boolean variable that can be set to `true` if a more verbose output is desired, useful for debugging the program.

The first list of methods (lines 27-32) implements the complete search procedures that can be called from an external environment. Each one corresponds to one of the methods we have explained in Chapter 4. In particular, `localSearch` is a simple steepest descent search, while `tabuSearch` adds a tabu list to the basic procedure.

The second list (lines 34-43) provides methods that can be called to interact with the model.

`setSolution`:	move to solution provided
`greedySolution`:	generate a solution with a Greedy method
`graspSolution`:	generate a solution with a GRASP method
`printSolution`:	print the current solution
`getSolutionArray`:	return the current solution as an array
`getObjective`:	return the cost of current solution
`getIterations`:	return the number of iterations performed up to now
`getElapsedTime`:	return the elapsed time up to now
`reset`:	reset the search a move to a random solution
`close`:	restore the best solution and print various data

The final set of methods (lines 45-53) are used by the class itself and represent the core of the local search. `update` is called after every move to update various structures of the class.

We now describe the most important methods.

```
1  void Heuristic::shake(int k, set{int} Variables, set{int} Domain) {
2    bool[,] forbid;
3    forbid = new bool[Sessions,Clusters] = false;
4    forall(d in 1..k) {
5      select(i in Variables, v in Domain: x[i] != v && !forbid[i,v] ) {
6        x[i] := v;
7        forbid[i,v] = true;
8      } onFailure {
9        break;
10     }
11   }
12   this.update();
13 }
```

Listing 5.8: `shake` method.

shake The `shake` procedure (Listing 5.8) performs a simple move `k` times: it selects a random session and move it into another cluster. We keep a local list that forbids to move a session back to its original cluster, something that could make the move useless. In this way, with `k` increasing, the difference between the starting and the final solution increases too.

firstImprovement The `firstImprovement` procedure (Listing 5.9) performs two nested loops. The inner one (`forall`) selects the first move that makes the objective function decrease. The outer one (`do while`) repeats the previous action until no improving move can be found. The condition `x[i] != v` forbids to evaluate useless moves that keeps solution unchanged.

bestImprovement `bestImprovement` (Listing 5.10) moves to the best solution in the neighborhood. Again, the action is repeated until no improving move can be found.

neighborhoodChange The `neighborhoodChange` procedure (Listing 5.11) makes a decision based on the current solution and the one received as a parameter (`Solution s`). If current solution is better, it increases `k` by one. Otherwise it moves to the original solution and reset `k` to 1.

findBest In the `findBest` method (Listing 5.12) three different neighborhoods are explored, according to the `k` received as parameter. What-

```
1  void Heuristic::firstImprovement(set{int} Variables, set{int} Domain) {
2    int oldValue;
3    do {
4      oldValue = obj.evaluation();
5      forall(i in Variables, v in Domain: x[i] != v ) {
6        if (obj.getAssignDelta(x[i],v) < 0) {
7          x[i] := v;
8          this.update();
9          break;
10       }
11     }
12   } while(obj.evaluation() < oldValue);
13 }
```

Listing 5.9: `firstImprovement` method.

```
1  void Heuristic::bestImprovement(set{int} Variables, set{int} Domain) {
2    int delta = 0;
3    int oldValue;
4    do {
5      oldValue = obj.evaluation();
6      selectMin(i in Variables, v in Domain,
7       delta = obj.getAssignDelta(x[i],v): x[i] != v ) (delta) {
8        if (delta < 0) {
9          x[i] := v;
10         this.update();
11       }
12     }
13   } while(obj.evaluation() < oldValue);
14 }
```

Listing 5.10: `bestImprovement` method.

```
1  int Heuristic::neighborhoodChange(Solution s, int oldValue, int k) {
2    int newValue = obj.evaluation();
3
4    if (newValue < oldValue) {
5      k = 1;
6    } else {
7      s.restore();
8      k++;
9    }
10
11   return k;
12 }
```

Listing 5.11: `neighborhoodChange` method.

ever the `k`, the execution performs a single move to the best solution in the neighborhood N'_k. Note how in N'_k the swap move is obtained easily with the built-in method `obj.getSwapDelta(x[i], x[l])`. With `k==3` however, it is necessary to explore manually the neighborhood and continuously restore the starting solution at each failure. To speed up this process, we to stop at the first improving solution obtained. The condition `!(v1 == x[l] && v2 == x[i])` ensures that we do not explore the region corresponding to a swap move: that region, in fact, cannot contain improving solution.

VND `VND` (Listing 5.13) is a descent method and, like `FirstImprovement` or `bestImprovement`, it presents a `do while` loop that stops when no improving move can be found. The inner `while` loop moves to the best solution by calling `findBest`, eventually increasing the size of the neighborhood to explore exhaustively. When even neighborhood `kMaxDescent` fails it stops.

unfeasibleSearch Since the code is quite long, some parts are substituted by a comment that summarizes them (Listing 5.14). At the beginning the method checks whether the received solution has already been visited by itself and exits in case of success. At the end the dual action is executed: the starting solution is added to the `visited` list. In lines 5-11 the method declares some variables needed next and it increases the domain of the search: the number of possible clusters is incremented by 1. Then, it performs a

```
1  void Heuristic::findBest(int k) {
2    int delta = 0;
3    switch (k) {
4      case 1:
5        selectMin(i in Sessions, v in Clusters,
6          delta = obj.getAssignDelta(x[i], v): x[i] != v) (delta) {
7          x[i] := v;
8        }
9      break;
10     case 2:
11       selectMin(i in Sessions, l in Sessions,
12         delta = obj.getSwapDelta(x[i], x[l]):
13         l > i && x[i] != x[l]) (delta) {
14           x[i] :=: x[l];
15       }
16     break;
17     case 3:
18       Solution original = new Solution(ls);
19       Solution best = new Solution(ls);
20       int bestValue = obj.evaluation();
21       bool stop = false;
22       forall(i in Sessions: !stop) {
23         forall(l in Sessions: l > i && !stop) {
24           forall(v1 in Clusters: v1 != x[i] && !stop) {
25             forall(v2 in Clusters: v2 != x[l] &&
26               !(v1 == x[l] && v2 == x[i]) && !stop) {
27               original.restore();
28               x[i] := v1;
29               x[l] := v2;
30               if (obj.evaluation() < bestValue) {
31                 best.refresh(ls);
32                 bestValue = obj.evaluation();
33                 stop = true;
34       }}}}}
35       best.restore();
36     break;
37   }
38   this.update();
39 }
```

Listing 5.12: `findBest` method.

```
1  void Heuristic::VND() {
2    Solution s = new Solution(ls);
3    int oldValue;
4    int cost;
5    int k = 1;
6    do {
7      oldValue = obj.evaluation();
8      while (k <= kMaxDescent) {
9        s.refresh(ls);
10       cost = obj.evaluation();
11       this.findBest(k);
12       k = this.neighborhoodChange(s, cost, k);
13     }
14   } while (obj.evaluation() < oldValue);
15 }
```

Listing 5.13: `VND` method.

simple descent method (whether `firstImprovement` or `bestImprovement`) in the new unfeasible space. When a local minimum is found two clusters are merged, obtaining a solution feasible for the original problem. To do this, it loops through all the possible pairs of clusters and selects the best pair, i.e. the one that increases the objective solution the least. Finally the labels of the clusters are reordered to fit in the original space and the domain is reduced to its original set.

VNS and VNUS The methods `VNS` and `VNUS` contain the complete search procedure and are the ones we used for the tests. They have been chosen, among all the variations, because they have shown a better performance in general: more details will be given in Chapter 6. Their code is similar, like other search methods, so only `VNS` is needed (Listing 5.15).

First the sets `Variables` and `Domain` are populated, as they are needed by the methods that performs the actual moves. Then `k` is initialized to 1. Next a `while` loop is executed. The loop iterates the search until `k` exceeds `kMax`, increasing `k` every time a local minimum is found. `VNUS` also adds the *space change* component at the end of each iteration.

```
1  void Heuristic::unfeasibleSearch(set{int} Variables, set{int} Domain) {
2
3    // Do not start if you receive the same previous solution
4
5    int extra = p + 1;
6    int delta;
7    Solution best = new Solution(ls);
8    int bestCost;
9    Solution bestUnfeasible = new Solution(ls);
10   int oldValue;
11   Domain.insert(extra);
12
13   // Perform a descent in the unfeasible space
14
15   bestUnfeasible.refresh(ls);
16   bestCost = n*m;
17   int empty;
18   forall(v1 in Clusters) {
19     forall(v2 in Domain: v2 > v1) {
20       bestUnfeasible.restore();
21       forall(l in Sessions: x[l] == v2) {
22         x[l] := v1;
23       }
24       if (obj.evaluation() < bestCost) {
25         best.refresh(ls);
26         bestCost = obj.evaluation();
27         empty = v2;
28       }
29     }
30   }
31   best.restore();
32   forall(i in Sessions: x[i] == extra) {
33     x[i] := empty;
34   }
35   Domain.delete(extra);
36   this.update();
37
38   // Remember this solution
39 }
```

Listing 5.14: `unfeasibleSearch` method.

```
1  void Heuristic::VNS(int mode) {
2
3    set{int} Variables();
4    set{int} Domain();
5    forall(i in Sessions) {
6      Variables.insert(i);
7    }
8    forall(v in Clusters) {
9      Domain.insert(v);
10   }
11
12   Solution s = new Solution(ls);
13
14   int cost;
15   int k = 1;
16   while (k <= kMax) {
17     s.refresh(ls);
18     cost = obj.evaluation();
19     this.shake(k, Variables, Domain);
20     if(mode == 1) this.firstImprovement(Variables, Domain);
21     if(mode == 2) this.bestImprovement(Variables, Domain);
22     if(mode == 3) this.VND();
23     k = this.neighborhoodChange(s, cost, k);
24   }
25
26 }
```

Listing 5.15: `VNS` method.

5.2.2 Column Generation

The declaration of the main class is shown in Listing 5.16.

```
1  class CG {
2
3    int n;
4    int m;
5    int p;
6    range Sessions;
7    range Clients;
8    range Clusters;
9    int[,] adj;
10
11   Solver<LP> lp;
12   stack{MasterVar} C;
13   range Variables;
14   bool integer;
15   int[] intSol;
16   float LB;
17   int[] prevCol;
18
19   CG(int _n, int _m, int _p, int[,] _adj, int[] sol);
20
21   void columnGeneration(int pricing);
22   float getLowerBound();
23   set{int} getBranchingVar();
24   bool isInteger();
25   int[] getIntegerSolution();
26   set{int} pricingMIP(float v, float[] u);
27   set{int} pricingCP(float v, float[] u, Boolean feasible);
28   set{int} pricingLS(float v, float[] u, Float reducedCost);
29
30 }
```

Listing 5.16: Class `CG`, for Column Generation.

Lines 3-9 declare the variables needed to store the instance of the problem, exactly like the `Heuristic` class. Lines 11-17 declare the linear programming solver and some utility variables to support the execution:

`Solver<LP> lp`:	the LP solver for the Master Problem
`stackMasterVar C`:	the list of variables for the RMP
`Variables`:	the range over the previous variables
`integer`:	true if the solution of RMP is integer
`intSol`:	the integer solution
`LB`:	the lower bound
`prevCol`:	the last generated column

With *lower bound* we mean the value of the objective function of the best solution found by the RMP, when the generation of columns is terminated. The list `C` contains objects of class `MasterVar`, shown in Listing 5.17.

```
1  class MasterVar {
2    var<LP>{float} lambda;
3    set{int} session;
4    int cost;
5    MasterVar (var<LP>{float} _lambda, set{int} _session, int _cost) {
6      session = _session;
7      lambda = _lambda;
8      cost = _cost;
9    }
10   var<LP>{float} lambda() { return lambda; }
11   bool contains(int i) { return session.contains(i); }
12   int getCost() { return cost; }
13   set{int} getElements() {return session.copy();}
14 }
```

Listing 5.17: `MasterVar` class.

The class represents a cluster and contains:

`lambda`:	an LP rational variable ranging in $[0:1]$
`session`:	the set of nodes contained in the cluster
`cost`:	the cost of the cluster

Back to class `CG`, we found the constructor of the class at line 19. It requires the instance data and an array of `int` containing the solution to start with: it is going to be required for generating the first columns for the RMP.

Finally the methods (lines 21-28) are declared. The following ones are called from an external environment to query information about the result of the process:

`getLowerBound`:	return the lower bound stored in `LB`
`getBranchingVar`:	return the variable suggested for branching
`isInteger`:	return `true` if the found solution is integer
`getIntegerSolution`:	return the integer solution

The "branching" variable can be chosen according to different criteria. We implemented a common choice, based of the value of the variable in the RMP, at the end of column generation: the variable with the most fractional value is returned. Different ideas could be experimented too.

```
1   lp = new Solver<LP>();
2   C = new stack{MasterVar}();
3   forall (v in Clusters) {
4     set{int} nodes();
5     forall (i in Sessions) {
6       if (sol[i] == v) {
7         nodes.insert(i);
8       }
9     }
10    int lambdaCost = sum(i in Sessions, j in Clients: adj[i,j] == 0)
11      ( ( sum(l in Sessions: adj[l,j] == 1)
12      (nodes.contains(i) && nodes.contains(l) ) ) >= 1);
13    C.push( MasterVar(new var<LP>{float}(lp), nodes, lambdaCost) );
14  }
15  Variables = 0..C.getSize()-1;
```

Listing 5.18: Extract from the constructor of class `CG`.

An extract of the constructor in shown in Listing 5.18. First the solver and the list of `MasterVar` are instantiated. Then the list is populated based on the starting solution. For each cluster the set of nodes contained in it is defined, the cost is computed and a new LP variable is created and registered to the LP solver. Finally the range `Variables` is defined over the list of clusters just created.

We now describe the most important methods.

columnGeneration The first lines of Listing 5.19 and 5.20 define the Master Problem (see Section 4.3). The first two families of constraints are declared. Then the objective function is expressed, followed by the definition of the constraints.

```
1  void CG::columnGeneration(int pricing) {
2
3    Constraint<LP> allClusters;
4    Constraint<LP> partition[Sessions];
5    minimize<lp>
6      sum(t in Variables) (C{t}.getCost() * C{t}.lambda())
7    subject to {
8      allClusters = lp.post(sum(t in Variables)(C{t}.lambda())==p);
9      forall (i in Sessions) {
10       partition[i] = lp.post(sum(t in Variables:
11        C{t}.contains(i)) (C{t}.lambda()) >= 0.999);
12     }
13   }
14
15   Float reducedCost(1.0);
16   while (true) {
17     float v = allClusters.getDual();
18     float u[i in Sessions] = partition[i].getDual();
19     Boolean feasible(true);
20     set{int} nodes();
21     switch (pricing) {
22       case 1:
23         nodes = this.pricingMIP(v, u);
24       break;
25
26       case 2:
27         nodes = this.pricingCP(v, u, feasible);
28       break;
29
30       case 3:
31           nodes = this.pricingLS(v, u, reducedCost);
32           if (reducedCost > -0.0001) {
33             nodes = this.pricingMIP(v, u);
34           }
35       break;
36     }
```

Listing 5.19: `columnGeneration` method (part 1).

```
37
38      int cost = sum(i in Sessions, j in Clients: adj[i,j] == 0)
39        ( (sum(l in Sessions: adj[l,j] == 1)
40        (nodes.contains(i) && nodes.contains(l)) ) >= 1);
41      float obj = cost - sum(i in Sessions: nodes.contains(i)) (u[i]) - v;
42
43      if (feasible && obj < -0.0001) {
44        Column<LP> col(lp);
45        col.setObjectiveCoefficient(cost);
46        col.setConstraintCoefficient(allClusters.getId(), 1);
47        forall(i in Sessions) {
48          if(nodes.contains(i)) {
49            col.setConstraintCoefficient(partition[i].getId(), 1);
50          } else {
51            col.setConstraintCoefficient(partition[i].getId(), 0);
52          }
53        }
54        var<LP>{float} gamma(lp, col);
55        C.push( MasterVar(gamma, nodes, cost) );
56        Variables = 0..C.getSize()-1;
57      } else {
58        break;
59      }
60    }
61
62    LB = lp.getObjective().getValue().getFloat();
63  }
```

Listing 5.20: `columnGeneration` method (part 2).

The column generation loop starts at line 16. The values of the dual variables are retrieved from the constraints and stored in `v` and `u`. Then the execution is switched according to the received parameter `pricing`. In the first case the pricing is solved using a MIP solver, in the second using a CP solver. When `pricing` is set to 3 the heuristic is executed: the pricing is solved with the local search and, when this method fails to find a column of negative reduced cost, an exact method is called (MIP in this case).

After the `switch` has performed an action, the cost of the found cluster and the reduced cost of the column are computed. If no column with a negative reduced cost could be found, the `while` loop is terminated (`break` of line 58). Otherwise, the column is added to the RMP and the coefficients of objective function and constraints are set. Then a new `MasterVar` object is added to the list `C`. The final line of the method simply stores the lower bound found.

pricingMIP The method contains a MIP model solved to optimality (Listing 5.21). The first lines declare the model:

`nodes`: the set of session nodes forming the cluster
`pr`: the MIP solver
`X`: the variable telling which sessions form the cluster
`Z`: the variable telling which edges we have to pay

Next the IP model is formulated: the objective function is the reduced cost of the cluster, while the constraints simply make the variables consistent among them. Finally the set of nodes is populated based on variable `X`.

pricingCP The CP model (Listing 5.22) is similar to the MIP one. The first difference consists in the absence of the objective function and the addition of one further constraint that impose to find a solution with a negative reduced cost. It becomes a satisfiability problem: it either finds a feasible solution or states that no such solution exists. The second difference is the control of the search tree: we first set to 1 the `X` variable corresponding to the nodes with highest dual multiplier `u`.

Finally, if a feasible solution has been found, the set of nodes is created, otherwise the boolean variable `feasible` is set to false.

pricingLS The first lines declare the model for the search (Listing 5.23). Only one variable `X` is needed to characterize a solution: it takes value 1 if the node is elected to form the cluster, exactly like the models of MIP and

```
1  set{int} CG::pricingMIP(float v, float[] u) {
2
3    set{int} nodes();
4    Solver<MIP> pr();
5    var<MIP>{int} X[Sessions] (pr, 0..1);
6    var<MIP>{int} Z[Sessions,Clients] (pr, 0..1);
7
8    minimize<pr>
9      ( sum(i in Sessions, j in Clients: adj[i,j] == 0) Z[i,j] ) -
10     ( sum(i in Sessions) (u[i] * X[i]) ) - v
11   subject to {
12     forall(i in Sessions, l in Sessions, j in Clients:
13       adj[i,j] == 0 && adj[l,j] == 1) {
14       pr.post( Z[i,j] >= X[i] + X[l] - 1 );
15     }
16   }
17
18   forall(i in Sessions) {
19     if (X[i] == 1) {
20       nodes.insert(i);
21     }
22   }
23   return nodes;
24 }
```

Listing 5.21: `pricingMIP` method.

```
1  set{int} CG::pricingCP(float v, float[] u, Boolean feasible) {
2
3    set{int} nodes();
4
5    Solver<CP> pr();
6    var<CP>{int} X[Sessions] (pr, 0..1);
7    var<CP>{int} Z[Sessions,Clients] (pr, 0..1);
8
9    solve<pr> {
10     pr.post(sum(i in Sessions, j in Clients: adj[i,j] == 0) (Z[i,j])
11       - sum(i in Sessions) (X[i] * u[i]) - v < -0.0001);
12     forall(i in Sessions, j in Clients: adj[i,j] == 0) {
13       forall(l in Sessions: l != i && adj[l,j] == 1) {
14         pr.post(Z[i,j] >= X[i] + X[l] - 1);
15       }
16     }
17   } using {
18     while(!bound(X)){
19       selectMax(i in Sessions: !X[i].bound()) (u[i]) {
20         try<pr> pr.label(X[i],1); | pr.label(X[i],0);
21       }
22     }
23   }
24
25   if (bound(X)) {
26     forall(i in Sessions) {
27       if (X[i] == 1) {
28         nodes.insert(i);
29       }
30     }
31     feasible := true;
32   } else {
33     feasible := false;
34     nodes = {};
35   }
36   return nodes;
37 }
```

Listing 5.22: `pricingCP` method.

```
1  set{int} CG::pricingLS(float v, float[] u, Float reducedCost) {
2
3    set{int} nodes();
4
5    Solver<LS> pr();
6    var{int} X[Sessions](pr, 0..1) := 0;
7    FloatFunction<LS> obj;
8    obj = new FloatFunctionExpr<LS>(sum(i in Sessions, j in Clients:
9     adj[i,j] == 0) ( (sum(l in Sessions: adj[l,j] == 1)
10    (X[i] == 1 && X[l] == 1)) >= 1) -
11    sum(i in Sessions) (X[i] * u[i]) - v);
12   pr.close();
13
14   forall(i in Sessions) {
15     if(prevCol[i]==0) {
16       X[i] := 1;
17     }
18   }
19
20   Solution s = new Solution(pr);
21   float cost;
22   int k = 1;
23   int kMax = n+1;
24   while (k <= kMax) {
25     s.refresh(pr);
26     cost = obj.evaluation();
27     // Shake
28     // Best Improvement
29     if(obj.evaluation() < -0.0001) {break;}
30     // Neighborhood Change
31   }
32
33   forall(i in Sessions) {
34     if (X[i] == 1) {
35       nodes.insert(i);
36     }
37     prevCol[i] = X[i];
38   }
39   reducedCost := obj.evaluation();
40   return nodes;
41 }
```

Listing 5.23: `pricingLS` method.

CP. The search is driven by the objective function `obj`, that computes the reduced cost of the cluster.

The starting solution is initialized in lines 14-18: given the solution generated at last iteration of the search, the opposite one is taken. This trick suggests the search to favor columns different from each other. Then a VNS procedure is executed. Since the code is very similar to the one explained in the previous section, it is substituted by comments. The only line worth mentioning if line 29: as soon as a local minimum has a negative reduced cost the search is terminated. This action drastically speeds up the generation of columns.

Finally, the set of nodes is populated, the solution is remembered for next method call and the reduced cost computed.

5.2.3 Branch-and-Price

The complete Branch-and-Price procedure is realized by the class `BranchPrice`, whose main attributes and methods can be read in the diagram of Figure 5.1. The first attributes declare the model of the problem, the other ones are:

`bestUB`:	the global best upper bound
`bestSol`:	the global best solution
`search`:	the local search object
`cg`:	the column generation object

The methods are:

`BranchPrice`:	constructor
`start`:	start the whole procedure
`depthFirst`:	perform a depth first strategy search
`getObjective`:	return the objective value of a solution
`getFeasible`:	find a feasible solution given the branching rules

The `start` procedure is called from an external environment and correspond to the root node of the B&B tree. At the end it calls the `depthFirst` method twice, one for each child. Since the steps performed by `start` are similar to `depthFirst`, just a subset of them, we only describe the latter.

To develop the complete procedure, the classes `Heuristic` and `CG` have been modified. In fact, now the branching rules must be considered in both the local search and the generation of columns. This has been achieved by adding the dimension of a node as a weight in the objective functions and a linear constraint for each forbidden pair imposed by the branching. However, no code will be shown, as nothing new is involved from the computational

point of view.

depthFirst Since the method is quite long and it contains many complicated details dealing with programming issues, the majority of the code will be omitted, leaving only the important parts (Listing 5.24 and 5.25).

Recall that the branching is done on the pair of nodes (i, j): the left child is obtained by forbidding that i and j stay in the same cluster; the right child is obtained by forcing i and j to be together in the same cluster. The first condition is ensured by keeping a list of pairs of nodes that cannot be put together, while the second one is achieved by creating a super-node containing both i and j.

The parameter requested by the methods are:

`_inst`: the instance of the problem
`_pair`: the branching pair, set by the father of the node
`_no`: the list of forbidden pairs of nodes
`_change`: whether we have to create a super-node with `_pair`

The `Instance` class will be discussed further on. `PairStack` is just a support class that abstracts a list of pairs of nodes. The local variables declared are:

`cluster`: the cluster returned by the column generation
`branch`: the pair in which we decide to branch
`inst`: the instance considered in this node
`ub`: the upper bound of this node
`lb`: the lower bound of this node
`sol`: a solution
`dead`: whether this node is dead and must be closed
`no`: the list of forbidden pairs of nodes

Lines 13-26 define the instance that will be used in this node: if `_change` is `true` we are in a right child and we need to generate a new instance, combining `_pair` into a super-node. Otherwise we just copy the received instance `_inst`. Then we generate a feasible solution to start with: this step is needed, since the branching rules stored in `no` are generally inconsistent with the solution of our father. Next, we check immediately if we have reached a particular type of leaf. This happens when the creation of super-nodes has lead to an instance with `p` nodes only: in that case, only one solution is allowed.

Subsequently the upper bound of the node is computed. A new `Heuristic` object is created, with the current instance and the branching rules. The

```
1  void BranchPrice::depthFirst(Instance _inst, set{int} _pair,
2    PairStack _no, bool _change) {
3
4    set{int} cluster();
5    set{int} branch();
6    Instance inst;
7    int ub;
8    float lb;
9    int sol[Sessions];
10   bool dead = false;
11   PairStack no();
12
13   if(_change) {
14     bool good;
15     inst = new Instance();
16     good = inst.combinePair(_inst, _pair);
17     if(!good) {
18       dead = true;
19     }
20     // Update PairStack no, according to the new instance
21   } else {
22     inst = new Instance(_inst);
23     forall(r in 0.._no.getSize()-1) {
24       no.push(_no.getPair(r));
25     }
26   }
27
28   if(!dead) {
29     sol = this.getFeasible(inst.getLeftSize(), no);
30     if (sol[1] == 0) {
31       dead = true;
32     }
33   }
34
35   if(!dead && inst.getLeftSize() == p) {
36     dead = true;
37   }
```

Listing 5.24: `depthFirst` method (part 1).

```
38
39    if(!dead) {
40      search = new Heuristic(inst, p, no, false);
41      search.setSolution(sol);
42      search.VNS(1);
43      sol = search.getSolutionArray();
44      ub = search.getObjective() + inst.getCost();
45    }
46
47    if(!dead) {
48      cg = new CG(inst, p, sol);
49      cg.setBranchingRule(no);
50      cg.columnGeneration(3);
51      lb = cg.getLowerBound() + inst.getCost();
52      cluster = cg.getBranchingVar();
53      if (ceil(lb) >= bestUB) {
54        dead = true;
55      } else if (cg.isInteger()) {
56        dead = true;
57        int[] intSol = cg.getIntegerSolution();
58        bestSol = inst.convertSolution(intSol);
59      }
60    }
61
62    // Decide the pair on which to do branching
63
64    if (!dead) {
65      no.push(branch);
66      this.depthFirst(inst, branch, no, false);
67      no.pop();
68      this.depthFirst(inst, branch, no, true);
69    }
70  }
```

Listing 5.25: `depthFirst` method (part 2).

object is requested to find a better solution through VNS (or any of the available methods). Then it is the turn of the lower bound. Similarly to the previous step, a new `CG` object is created. The lower bound is then stored and two conditions are checked. First we close the node if the lower bound is worse than the global best upper bound. Secondly, if the relaxation obtained by column generation is integer, we have found the optimum of the problem.

Next a pair of nodes must be elected to do the branching and the generation of two children. However this step is delicate, as it is not possible to create an inconsistent situation with respect to the previous branching rules received by the ancestors. Basically a pair is selected from the fractional cluster returned by the Column Generation procedure. If no pair could be find, than a random one is chosen. If every thing fails, then a leaf has been reached and the node is closed.

Finally, if all the previous steps have been correct, the two children are generated by calling `depthFirst` twice. The first child receives an additional pair in the list `no`, while the second one is asked to generate a new instance with the received pair `branch`. The methods `push` and `pop` allow to insert a pair in the list for the first child and to remove it immediately after it terminates, so that the second child is free from that rule.

The class `Instance`, used in the Branch-and-Price, is shown in Listing 5.26. The attributes are:

`n`:	size of left shore
`m`:	size of right shore
`adj`:	set of edges
`map`:	mapping needed during transformation
`nodes`:	content of each super-node
`dim`:	cardinality of each super-node
`cost`:	implicit cost of the instance

Then we find the constructors. Next we find all the methods. The first ones are straightforward, only the last ones need a brief explanation. The class `Nodes` simply abstracts a set of nodes.

`getFamily` returns the set of nodes that are in the same super-node of `_node`.

`combinePair` changes the internal structure of the objects. It creates a super-node from the received instance `original` by merging the two nodes contained in `_pair`. All the variables are modified accordingly.

```
1  class Instance {
2    int n;
3    int m;
4    int[,] adj;
5    dict{int->int} map;
6    dict{int->Nodes} nodes;
7    int[] dim;
8    int cost;
9
10   Instance();
11   Instance(int _n, int _m, int[,] _adj);
12   Instance(Instance _inst);
13
14   int getLeftSize() {return n;}
15   int getRightSize() {return m;}
16   int[,] getEdges() {return adj;}
17   int getCost() {return cost;}
18   int[] getDim() {return dim;}
19   Nodes getNodeContent(int key) {return nodes{key};}
20   Nodes getFamily(int _node);
21   bool combinePair(Instance original, set{int} _pair);
22   int[] convertSolution(int[] sol);
23 }
```

Listing 5.26: Class `Instance`.

It returns false in case of error.

`convertSolution` returns an array of `int` expressing solution `sol` in terms of the original nodes, without super-nodes. It maintains the correspondence between a solution in the current instance and the same solution in the original problem.

Chapter 6

Results

As stated at the beginning of the document, extensive testing is of fundamental importance in the study of optimization methods. In this chapter the most significant results are presented. Our goal is to show how the implementation discussed in Chapter 5 effectively allows to solve the k-CmBCP, both approximately and exactly. We also want to underline which choices proved to be correct and which did not, what are the limits in the algorithms and what instances constitute better candidates for good results.

First we are going to describe what kind of data has been used. Then the remainder of the chapter is dedicated to the results of the tests.

We list here the notations that will be used in the following tables and charts. Many of them have been used consistently throughout the document, but we recall everything here for clarity.

The parameter **d** is used for random generated instances only, while **model** is used to distinguish among different mathematical programming formulations. When time will not be expressed in seconds, the unit measure will be explicitly written. A '-' symbol indicates that data is not available: generally it is because the execution reached a time out before terminating. The gap between the found solution s and the optimal one $s^\star$ is defined as:

$$\textbf{Gap} = \frac{|s - s^\star|}{s^\star}$$

The tests presented here have been executed on a machine with a Intel(R) Xeon(TM) 2.80GHz CPU and 2GB of memory. For space issues, only a small fraction of the numerous experimental tests are reported, but the missing ones are quantitatively similar and bring to the same conclusions.

Table 6.1: Notations used in this chapter.

|I|	Number of nodes in the left shore.
|J|	Number of nodes in the right shore.
d	Density of edges in the graph.
k	Number of clusters in which nodes must be partitioned.
model	Name of the model subject to the test.
Time	Time required to run the test, usually in seconds.
It.	Iterations executed.
Opt	Value of the optimal solution.
LB	Lower Bound.
UB	Upper Bound.
Gap	Percentage distance of found solution w.r.t. optimal solution.
LS	Local Search heuristic.
TS	Tabu Search heuristic.
VNS	Variable Neighborhood Search heuristic.
VNUS	Variable Neighborhood Unfeasible Search heuristic.
CG	Column Generation algorithm.
BP	Branch-and-Price algorithm.
Bn	Number of nodes generated during Branch-and-Price.

6.1 Instances

The data used for testing can be divided into two groups: the random instances and the ones coming from real collected data.

6.1.1 Random instances

Generating random instances is a necessity if one wants to test an algorithm on every dimension desired. Two methods have been used to generate random graphs.

The first method is quite straightforward. Given the number of nodes of the two shores and a probability d (density), we assign an edge between two nodes based on d. We have spent most of the testing on 3 probabilities: 0.3, 0.5, 0.7. The higher the probability, the more dense is the graph. The sizes of the shores range from 10 to 100, while the number of clusters required range from 2 to 8. Note that the size of an instance cannot be expressed simply by one number. It can be associated to the number of possible edges,

that is $n \times m$, but the number of clusters p must be taken into account too. That is why for each test we define the instance by all its attributes (n, m, p and d). Tests will show how different methods respond differently to the variation of some attributes. However we can refer to the size of an instance with n-m, when the other attributes can be ignored.

The second method goes a little further: it generates the instances while knowing exactly the optimal solution. In this way a test performed with a heuristic provides its precision (gap) too, without requiring an exact method to prove it. The basic idea is to construct the graph by using structures that preserve the optimality of the solution. The instances we have been able to generate in this way are of one type only: optimal clusters (quasi-bicliques) are completely disjoint subgraphs and the density of edges is always greater than 0.5. The reliability of the tests done on these type of instances is limited, since an algorithm could respond well to a particular type of instances, while performing very badly on other ones. However, this method represents the only possibility available for high dimension instances. Nodes on both shores have been shuffled, in order to avoid having clusters made of adjacent nodes only. Figure 6.1 shows an example of an instance with two disjoint quasi-bicliques: the left graph is the original instance, the right one is the same instance after the nodes have been shuffled. Optimal clusters defined of left shore are $\{1, 2\}$ and $\{3, 4\}$.

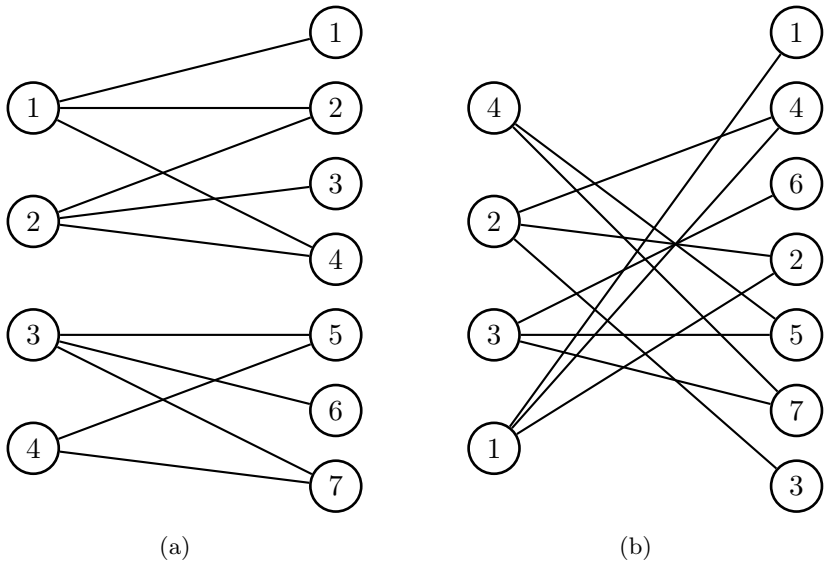


Figure 6.1: An example of instance with two disjoint quasi-bicliques: (b) is the same instance of (a) after node shuffling.

6.1.2 Real instances

To test the solution methods on realistic data we use the MovieLens data sets. They were collected by the GroupLens Research Project ([34]) at the University of Minnesota through the MovieLens web site ([35]) during the seven-month period from September 19th, 1997 through April 22nd, 1998. The data set that we are going to use consists of ratings on movies provided by users. It contains 100,000 ratings from 943 users on 1682 movies, ranging from 1 to 5. Each user has rated at least 20 movies and simple demographic info for the users are reported.

The dataset can be translated easily to our problem. Using the MPP as a reference, the users are the clients, while the movies are the sessions. A rating can be seen as a preference expressed by the client/user about the session/movie. In terms of k-CmBCP, users constitute the left shore, the items the right shore and the ratings form the edges.

The collection of data and the resulting graph are quite big. However, since the ratings are evenly spread, it has been possible to extract meaningful subgraphs to use for the tests. Generally the extracted sub-graphs are very sparse, that is the density of edges is low (less than 0.3).

6.2 IP formulations

In this section we investigate the goodness of the mathematical models. The solver used is ILOG CPLEX® ([36]) version 11. The models have been written in AMPL. These first results have also the objective to place a standard of performance against which our program could be compared. ILOG CPLEX®, in fact, is considered the fastest commercial MIP and LP solver (according to [37]).

Solving time

We start by listing the time needed by CPLEX® to solve the instances to optimality. All tests have a time out limit of 1000 seconds. Table 6.2 shows the data obtained in the tests with randomly generated graphs.

Formulation **1d** has proved to be the best overall. Models (model-1b) and (model-1c) are not reported, because the failed to solve even some instances of size 12-12. The table has shown that instances of size 18-18 are the maximum that can be solved with CPLEX® in the time limit, but it

Table 6.2: Comparison of the solving time of the IP models on random graphs.

				model		
|I|	**|J|**	**k**	**d**	**1d**	**2**	**3**
15	15	2	0.3	0.40	24	81
15	15	2	0.5	0.27	23	61
15	15	2	0.7	0.25	14	6.54
15	15	3	0.3	5.52	41	720
15	15	3	0.5	4.15	38	138
15	15	3	0.7	2.63	39	48
15	15	4	0.3	17	237	-
15	15	4	0.5	10	247	145
15	15	4	0.7	11	539	36
15	15	5	0.3	99	657	-
15	15	5	0.5	25	960	83
15	15	5	0.7	113	2764	54
18	18	2	0.3	3.15	173	545
18	18	2	0.5	0.53	54	252
18	18	2	0.7	0.56	80	70
18	18	3	0.3	34	942	-
18	18	3	0.5	14	651	-
18	18	3	0.7	12	807	706
18	18	4	0.3	228	-	-
18	18	4	0.5	161	-	-
18	18	4	0.7	88	-	-
18	18	5	0.3	-	-	-
18	18	5	0.5	991	-	-
18	18	5	0.7	380	-	-

also highlights how the density and, more heavily, the number of clusters influence the performance.

In table 6.3 the all the models solvable by CPLEX® are applied to the dataset. A subgraph containing students and animation movies has been extracted and some elements have been pruned further on to reach an in keeping size.

Table 6.3: Comparison of the solving time of the IP models on real graphs.

			model				
|I|	**|J|**	**k**	**1b**	**1c**	**1d**	**2**	**3**
12	12	2	0.15	0.09	0.17	3.70	2.03
12	12	3	4.98	2.72	0.63	3.96	2.44
12	12	4	266	-	0.35	3.25	1.63
12	12	5	-	-	0.42	0.99	1.23
15	15	2	0.25	0.16	0.53	14	5.79
15	15	3	65	25	3.24	8.17	12
15	15	4	-	-	4.99	12	7.55
15	15	5	-	-	5.12	11	4.09
18	18	2	0.22	0.17	1.16	10	9.04
18	18	3	224	175	1.21	3.18	56
18	18	4	-	-	2.96	6.03	49
18	18	5	-	-	2.29	11	18
20	20	2	0.26	0.13	1.08	21	15
20	20	3	251	235	2.54	17	112
20	20	4	-	-	7.62	22	92
20	20	5	-	-	7.73	26	129

Results show how real instances are easier to solve. This fact is explained by the great sparsity of the graphs and the presence of particular structures in the data. It is common, in fact, to have some movies that have received many ratings from similar users and, thus, are likely to be put in the same cluster.

Impact of changes in the instance structure

The goal of the following charts is to understand how the performance of the IP models is affected by each attribute characterizing an instance.

The first analysis can be made on the number of clusters. The bipartite

graph is always the same, but the value p is varied from 2 to 5. The average of the times needed to solve the instances is taken. In Figure 6.2 the values have been scaled to the highest values for each model.

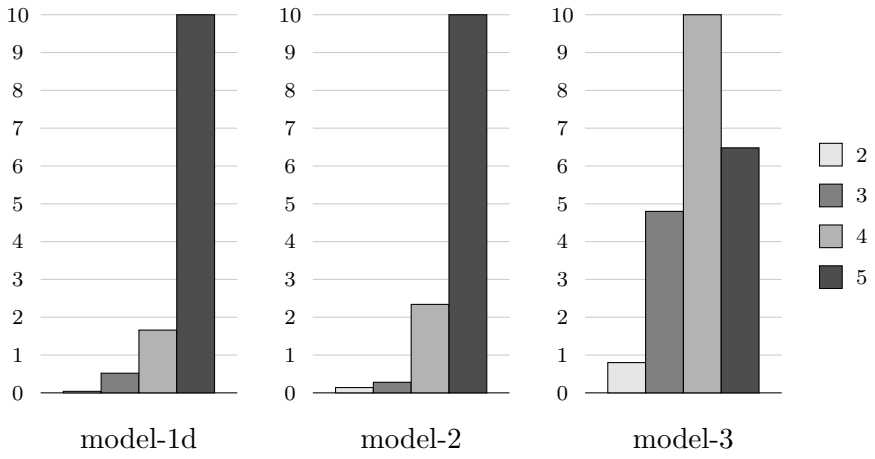


Figure 6.2: Impact of the number of clusters on the solution time required by the IP models.

For the rest of the tests p will be fixed to 4. Next chart (Figure 6.3) shows the average change in solving time in function of the density d of the edges. Again, values are relative to the highest one and independent among the models.

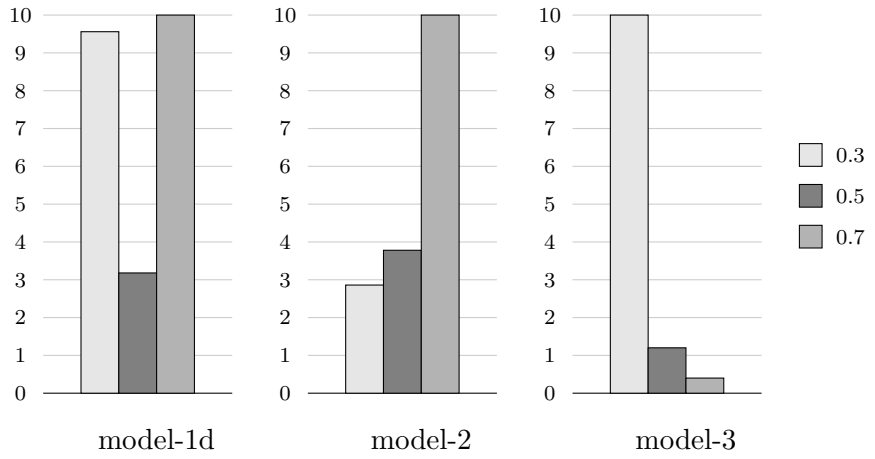


Figure 6.3: Impact of the density of edges on the solution time required by the IP models.

The previous figures have shown how strong is the impact of the at-

tributes of an instance and why they cannot be ignored. Furthermore, models expose a different behavior. (model-3) is more consistent than the other ones with respect to the number of clusters. It is the only one, in fact, that does not suffer from the problem of symmetries that arise when the order of the clusters is considered. The first two models have a number of equivalent solutions that grows with the number of cluster, hence the explanation of these results. Considering Figure 6.3, (model-3) shows to be heavily affected by the density of the edges, with the sparse instances requiring a lot more time to be solved. The reason can be found in the high number of z variables and their relative constraints when there are many $\{i, j\} \in \bar{E}$. The other models show some differences, but not as evident as (model-3).

6.3 Local Search heuristic

In this section the Local Search heuristic is tested as a stand-alone solution method. Among the several variations we focus our attention to the VNS with Best-Improvement strategy and the VNUS with First-Improvement as descent method. We refer to them as VNS-B and VNUS respectively. Furthermore, the basic local search and the tabu search are reported, in order to have a comparison. The reason we use two variations is that VNUS has proved to be the most capable of reaching a low-value solution, but VNS-B is useful to understand how much the *Unfeasible Search* improves the procedure. VNUS, in fact, is basically a VNS with an added search in the unfeasible space. Furthermore, in the Branch-and-Price, tested at the end of this chapter, VNS will be used in the nodes to compute the upper bound, with the exception of the root node that executes a VNUS.

In local search algorithms, the random component cannot be ignored. For this reason the tests have been repeated two times or more and the average has been taken as final result: the value of the objective function has been rounded to the first smaller integer.

Best solutions

We shall limit the search to one complete cycle only. For the simple local search this means to stop at the first local minimum. For the VNS variations it means to stop when all the k_{max} neighborhoods have been explored without success.

Table 6.4 compares the best solution found by the four heuristics for random instances. The table is used to highlight the ability of the algo-

rithms to reach the best solution possible. The two VNS variations perform a complete cycle, the local search stops and terminates at the first local minimum, while the tabu search runs for a time frame comparable with the one required by the VNS. Many dimensions of the shores, ranging from 15 to 80, have been considered and the average has been taken. The values are relative, scaled on the best values found by any of the methods.

Table 6.4: Relative value of the best solution found by the heuristics.

k	**d**	**LS**	**TS**	**VNS-B**	**VNUS**
2	0.3	1.04	1.02	1	1
2	0.5	1.07	1.03	1.01	1
2	0.7	1.09	1.03	1.07	1
3	0.3	1.06	1.03	1	1
3	0.5	1.07	1.02	1.01	1.01
3	0.7	1.13	1.04	1.02	1.02
4	0.3	1.03	1.02	1	1
4	0.5	1.06	1.02	1	1
4	0.7	1.10	1.03	1.01	1
5	0.3	1.07	1.03	1.03	1
5	0.5	1.04	1.02	1	1
5	0.7	1.07	1.02	1.01	1

Results show a better performance of the VNS variations with respect to Local and Tabu Search. VNUS is the best method overall, finding the best value on the majority of the instances.

Table 6.5 shows some hard instances of 50 nodes per shore, with a fixed time limit of 1000 seconds.

These last results confirm the conclusions taken from the previous table and they serve to ensure that, even with great amount of time at their disposal, TS and VNS-B generally fails to reach the same solution of VNUS (with the exception of one instance, for which the gap is low anyway).

Given the previous considerations, we now consider VNUS only. Table 6.6 shows its performance. We show whether the heuristic was able to find the optimum, when solving instances of problems for which the optimum is known. The table columns are: the size of left shore, the size of right shore, the time needed to reach the best solution, the number of iterations needed to reach the best solution, the value of the optimum, the value of the best

Table 6.5: Value of the best solution found by the heuristics in 1000s for instances of 50 nodes per shore.

k	d	TS	VNS-B	VNUS
3	0.3	1592	1500	1500
3	0.5	1165	1165	1153
3	0.7	728	740	713
4	0.3	1458	1407	1405
4	0.5	1134	1118	1108
4	0.7	697	702	692
6	0.3	1276	**1232**	**1237**
6	0.5	1058	1042	1028
6	0.7	674	662	652

solution found and the gap achieved by the algorithm.

VNUS has reached the optimum solution in all the presented instances for which optimum is known. It has proved to scale well with the dimension of the instances. The number of iterations increases at a reduced pace with the dimension of the instance, but the time per iteration must be considered too.

Table 6.7 shows similar results when VNUS is applied to real instances. The optimal solution is always found and in a very short time. These results, considered with Table 6.3, suggest that the real instances extracted from the Movielens database are particular easy to solve: the sparsity of the graph and the presence of high degree nodes are easily exploited by our local search heuristic.

Disjoint clusters instances

Table 6.8 summarizes the experiments performed on disjoint clusters instances, for which cost is known during generation (see Section 6.1.1).

The table clearly shows how the special case of disjoint clusters is solved easily to optimality by the VNUS.

Impact of changes in the instance structure

The following tests are on the same line as the ones presented before for the IP formulations, so the same considerations apply.

Table 6.6: Performance obtained on random graphs by VNUS with $p = 4$.

|I|	**|J|**	**d**	**Time**	**It.**	**Opt**	**Sol**	**Gap**
15	15	0.3	0.04	23	66	66	**0**
15	15	0.5	0.12	47	59	59	**0**
15	15	0.7	0.08	45	46	46	**0**
18	18	0.3	0.12	40	112	112	**0**
18	18	0.5	0.35	101	96	96	**0**
18	18	0.7	0.56	138	71	71	**0**
20	20	0.3	0.08	22	145	145	**0**
20	20	0.5	0.11	24	138	138	**0**
20	20	0.7	0.25	48	92	92	**0**
25	25	0.3	0.22	42	-	245	-
25	25	0.5	0.29	31	-	224	-
25	25	0.7	0.23	7	-	150	-

Table 6.7: Performance obtained on real graphs by VNUS.

|I|	**|J|**	**k**	**Time**	**It.**	**Opt**	**Sol**	**Gap**
15	15	2	0.02	35	73	73	**0**
15	15	3	0.01	14	49	49	**0**
15	15	4	0.01	12	32	32	**0**
15	15	5	0.17	148	22	22	**0**
18	18	2	0.01	8	101	101	**0**
18	18	3	0.06	40	54	54	**0**
18	18	4	0.09	74	35	35	**0**
18	18	5	0.01	17	23	23	**0**
20	20	2	0.01	13	124	124	**0**
20	20	3	0.09	29	77	77	**0**
20	20	4	0.01	20	47	47	**0**
20	20	5	0.11	40	34	34	**0**

Table 6.8: Performance obtained on disjoint clusters instances by VNUS.

|I|	**|J|**	**k**	**Sol**	**Opt**	**Gap**
20	20	2	2	2	0
20	20	3	7	7	0
20	20	4	9	9	0
30	30	2	3	3	0
30	30	3	11	11	0
30	30	4	15	15	0
30	30	5	11	11	0
40	40	2	3	3	0
40	40	3	14	14	0
40	40	4	14	14	0
40	40	5	27	27	0
50	50	2	3	3	0
50	50	3	16	16	0
50	50	4	30	30	0
50	50	5	19	19	0

In Figure 6.4 the impact of clusters (left) and density (right) is shown. The method tends to behave consistently well under any of the densities tested. Also the number of cluster does not affect the performance as much as in the IP models.

Objective function

The following plots represent the value of the objective function during the execution of the program. This type of analysis is useful to understand the behavior of an algorithm. It may happen, in fact, that the objective function lowers very quickly, but it ceases to lower any further very soon: all the rest of the time is wasted. Even though local search methods typically show a trend similar to the one just explained, different curves can be compared and extreme cases should be investigated deeply. Generally adding diversification during advanced steps of the execution can help in finding a better solution.

In the analysis, two different units can be assigned to the abscissa: time or iterations. In the first case the chart tells the absolute performance of the program. The second case gives more insight into the behavior of the

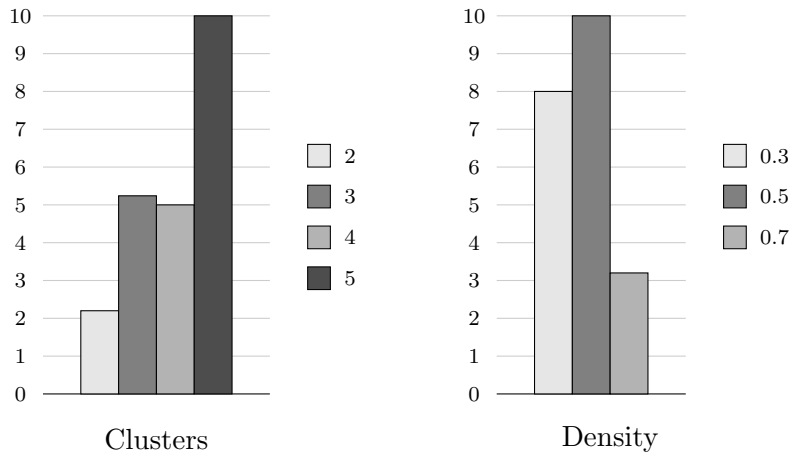


Figure 6.4: Impact of the number of clusters (left) and the density (right) on the performance of VNUS.

method, exposing the length of the steps too. A method, for instance, can perform less and slower iterations with respect to another one, but it can still achieve greater performance if it performs better moves inside each iteration.

We analyze the instance 30-30, with density 0.5 and number of clusters 4. In Figure 6.5 and 6.6 VNUS is compared to the Tabu Search.

The starting solution has been randomly generated for each execution: VNUS, in fact, starts at a worse solution. Figure 6.5 shows a steepest curve in the case of VNUS. Tabu search finds its best solution earlier, but it stops there, while VNUS has a longer time frame in which solution is improved. Figure 6.6 shows how VNUS iterations are faster: it requires more iterations to reach the same value of Tabu search. This is explained by the fact that Tabu Search selects the best solution at each iteration, while VNUS implements a First Improvement descent method.

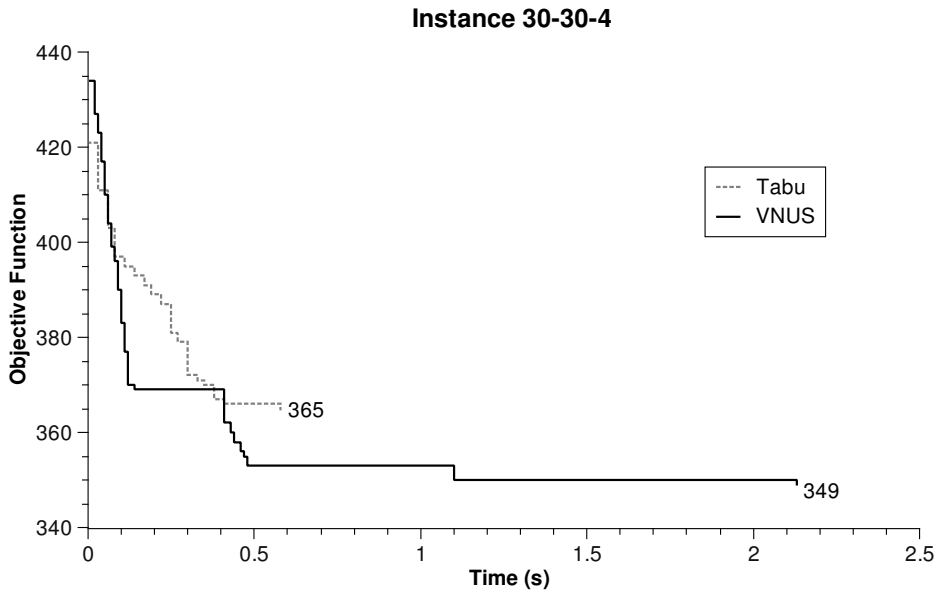


Figure 6.5: Comparison of the objective function values with respect to time.

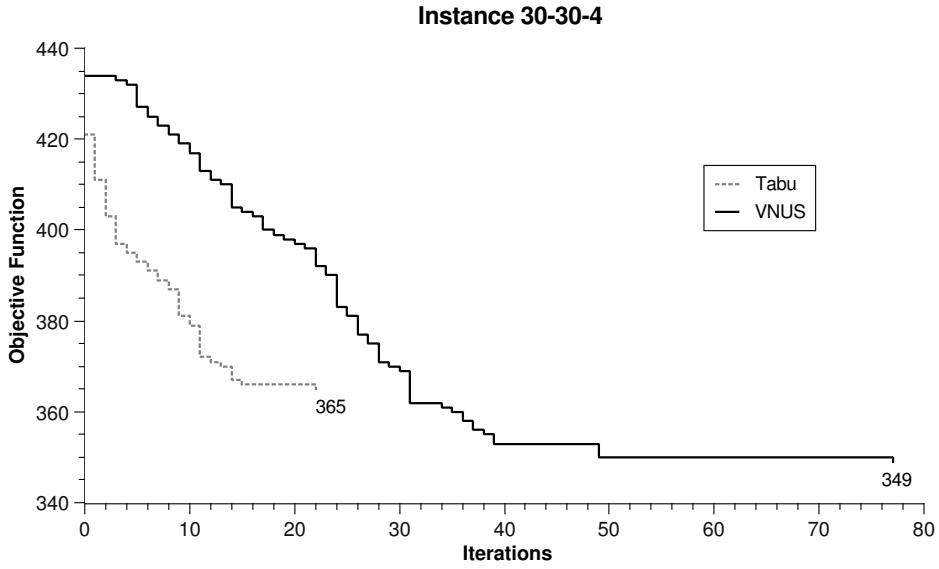


Figure 6.6: Comparison of the objective function values with respect to iterations.

6.4 Column Generation

In this section the Column Generation with the heuristic for the Pricing Problem is tested as a stand-alone relaxation method. In all the tests the starting columns of the RMP are created from the solution obtained by the VNUS heuristic. No other columns have been added, in order to analyze the performance of the algorithm when it has to generate all the columns by itself. It is possible though to keep more than one solution from the LS heuristic, and to add all the corresponding columns at once, before even starting the CG.

Solving time, with phases subdivisions

Table 6.9 shows the time needed to generate all the columns on random graphs. The first set of columns describes the instance. The second set refers to the whole Column Generation procedure. The third set contains details about the heuristic used to solve the Pricing and last set contains details about the Exact method. **Mean** is the average time required by an iteration.

The table shows a consistent behavior of the global method for every instance. The impact of the number of clusters and the density of the graph is almost negligible. Analyzing the data CG shows an opposite trend when it comes to the number of clusters: higher values of k requires less time to generate all columns.

The interesting results here concern the partition of time between heuristic and exact solution of the Pricing Problem. The heuristic generates the majority of the columns and requires an exact method for the last iterations only. However solving exactly the Pricing Problem is very time consuming and affects the total time by a non-negligible factor.

Note that the time required to solve the Pricing Problem by the heuristic is affected not only by the local search. Comet, in fact, adds a time overhead at each execution of the heuristic (each generation of a column) that cannot be controlled or eliminated by the developer and it is caused by a missing feature that is yet to be implemented. This overhead accounts for the majority of the time required at each generation of a column. For consistency with the rest of the tests, we have kept the whole time anyway.

Table 6.9: Time required for a complete Column Generation procedure, divided between heuristic and exact phases.

Instance				**Global**		**Heuristic**			**Exact**		
|I|	**|J|**	**k**	**d**	**Time**	**It.**	**Time**	**It.**	**Mean**	**Time**	**It.**	**Mean**
15	15	3	0.3	14	62	9.96	60	0.166	4.54	2	2.27
15	15	3	0.5	15	64	10	60	0.18	5.02	4	1.255
15	15	3	0.7	11	70	10	67	0.149	1.51	3	0.503
15	15	4	0.3	11	49	7.25	47	0.154	4.01	2	2.005
15	15	4	0.5	12	56	9.56	53	0.18	2.69	3	0.896
15	15	4	0.7	10	68	8.96	66	0.135	1.09	2	0.545
15	15	5	0.3	8.09	35	5.63	33	0.17	2.42	2	1.21
15	15	5	0.5	7.95	35	5.96	32	0.186	1.93	3	0.643
15	15	5	0.7	6.98	38	5.19	33	0.157	1.74	5	0.348
18	18	3	0.3	83	94	30	92	0.334	52	2	26.485
18	18	3	0.5	73	104	55	100	0.555	17	4	4.375
18	18	3	0.7	47	111	39	105	0.372	8.57	6	1.428
18	18	4	0.3	38	71	26	70	0.384	11	1	11.46
18	18	4	0.5	54	76	34	72	0.48	19	4	4.842
18	18	4	0.7	32	92	24	87	0.286	7.21	5	1.442
18	18	5	0.3	40	80	28	78	0.36	12	2	6.29
18	18	5	0.5	58	88	40	85	0.478	18	3	6.013
18	18	5	0.7	29	76	22	68	0.332	6.32	8	0.79
20	20	3	0.3	145	109	70	106	0.662	75	3	25.033
20	20	3	0.5	146	147	117	145	0.812	28	2	14
20	20	3	0.7	171	134	127	124	1.029	44	10	4.406
20	20	4	0.3	83	88	55	87	0.642	27	1	27.52
20	20	4	0.5	140	134	114	132	0.864	26	2	13.105
20	20	4	0.7	101	111	93	109	0.861	7.24	2	3.62
20	20	5	0.3	182	74	155	72	2.16	27	2	13.585
20	20	5	0.5	118	92	92	89	1.041	25	3	8.63
20	20	5	0.7	131	148	107	139	0.776	23	9	2.617

Quality of bounds

Table 6.10 shows the bound obtained by the CG with respect to the one provided by the VNS heuristic. Optimum value is shown, when available.

Table 6.10: Comparison of the bounds obtained by CG and VNUS with the optimum.

|I|	**|J|**	**k**	**d**	**LB**	**Time**	**UB**	**Time**	**Opt**
15	15	2	0.3	108	14	113	0.13	113
15	15	2	0.5	90	15	92	0.02	92
15	15	2	0.7	**61**	15	**61**	0.01	61
15	15	3	0.3	82	14	84	0.04	84
15	15	3	0.5	**72**	15	**72**	0.03	72
15	15	3	0.7	51	11	53	0.08	53
15	15	4	0.3	64	11	66	0.04	66
15	15	4	0.5	**59**	12	**59**	0.12	59
15	15	4	0.7	45	10	46	0.08	46
15	15	5	0.3	**51**	8.09	**51**	0.01	51
15	15	5	0.5	49	7.95	50	0.01	50
15	15	5	0.7	**39**	6.98	**39**	0.27	39
18	18	2	0.3	169	76	179	0.74	179
18	18	2	0.5	**124**	58	**124**	0.11	124
18	18	2	0.7	**89**	45	**89**	0	89
18	18	3	0.3	135	83	137	0.1	137
18	18	3	0.5	**109**	73	**109**	0.25	109
18	18	3	0.7	**79**	47	**79**	0.1	79
18	18	4	0.3	110	38	112	0.12	112
18	18	4	0.5	**96**	54	**96**	0.35	96
18	18	4	0.7	**71**	32	**71**	0.56	71
18	18	5	0.3	**92**	40	**92**	0.17	-
18	18	5	0.5	84	58	86	0.02	86
18	18	5	0.7	**63**	29	**63**	0.06	63

VNUS has reached the optimum in all the instances reported. The lower bound provided by CG is tight and on nearly half of the instances is equal to the upper bound. The table also shows how CG is a much slower procedure than VNUS. For this conclusion, the implementation overhead of the CG that we have mentioned previously must be taken into account.

6.5 Branch-and-Price

In this section the complete B&P procedure is tested. Given the previous tests, it is interesting now to see how fast can the B&P solve random instances with the implemented CG and LS heuristic.

Solving time

Table 6.11 shows the performance of the B&P. Other than the total time spent, further columns are reported:

UB the upper bound found at the root node by VNUS

LB the lower bound found at the root node by CG

Opt the optimum found the the B&P

Bn the number of nodes explored

LS time spent by the LS heuristic (global)

CG time spent by the CG (global)

Time total time

CPLEX time required by CPLEX to solve the instance

If **Bn** presents the value 0, it means that the execution terminated at the root node. The comparison is made against the most efficient IP model (model-1d).

As the dimension increases, the B&P tends to outperform CPLEX® in solving the same instance to optimality. Many instances requires no nodes to be explored, thanks to the goodness of the bounds found at the root node. Even when the bounds are different, the number of nodes remains low. It can be noted that CG accounts for the majority of time spent by the algorithm, underlining how critical it is. Note that in some instances the VNUS executed at root node fails to find the optimal solution. As this may happen considering the random factor involved, the main reason is because the LS heuristic has been given a fixed time to find a bound at each node. We have not tried to optimize the allowed time, since the goal here has been to show the behavior of the whole B&P procedure.

Table 6.11: Performance of the Branch-and-Price method compared with the IP model *1d* solved by CPLEX®.

|I|	**|J|**	**k**	**d**	**UB**	**LB**	**Opt**	**Bn**	**LS**	**CG**	**Time**	**CPLEX**
15	15	3	0.3	84	82	84	6	1.83	68	69	5.52
15	15	3	0.5	72	72	72	0	1.27	11	13	4.15
15	15	3	0.7	53	52	53	2	1.06	25	26	2.63
15	15	4	0.3	66	65	66	2	1.3	27	28	17
15	15	4	0.5	59	59	59	0	1.16	10	11	10
15	15	4	0.7	46	45	46	8	1.68	57	58	11
15	15	5	0.3	51	51	51	0	1.43	7.43	**8.86**	99
15	15	5	0.5	50	50	50	0	1.36	6.17	**7.53**	25
15	15	5	0.7	40	39	39	2	1.59	17	**19**	113
18	18	3	0.3	**138**	136	**137**	4	5.17	350	356	34
18	18	3	0.5	109	109	109	0	1.95	33	35	14
18	18	3	0.7	79	79	79	0	1.02	23	24	12
18	18	4	0.3	**113**	111	**112**	6	8.86	324	333	228
18	18	4	0.5	96	96	96	0	2.26	35	**37**	161
18	18	4	0.7	71	71	71	0	1.32	13	**15**	88
18	18	5	0.3	92	92	92	0	4.48	41	**45**	-
18	18	5	0.5	86	85	86	4	6.45	265	**271**	991
18	18	5	0.7	63	63	63	0	2.24	12	**14**	380
20	20	3	0.7	104	104	104	0	1.85	56	58	16
20	20	4	0.7	92	92	92	0	2.25	68	**70**	158
20	20	5	0.5	123	121	123	4	16	755	**772**	-
20	20	5	0.7	82	82	82	0	2.53	41	**44**	-

Chapter 7

Conclusions

The work presented in this document focused on the modelization of a particular biclique completion problem and the development of effective solution methods. The application field that brought the subject into light is Telecommunications, in particular the aggregation of multicast sessions. The problem, however, has other applications too, like the data mining examples we have mentioned in Chapter 2.

Studies of the problem, called k-CmBCP, had already been done before the beginning of this work. Some models and their properties had been formalized and some standard solution methods had been tested. What we have done here, have been a deeper analysis of the models and the development of ad-hoc algorithms, allowing to solve instances of higher dimension to optimality and very large instances to good sub-optimal solutions. In order to achieve our goals we have proceeded according to the following schedule:

- First it has been necessary to study and summarize the results found in the literature correlated with the problem. In parallel, a survey has been done to identify eligible algorithms and meta-heuristics. This step has allowed to identify the areas that could present the greatest difficulties.
- A modelization phase has followed, by means of combinatorial and integer programming formulations. The most important properties of each model have been outlined.
- The first step towards the solution of the k-CmBCP has been the design of a heuristic, with the goal of finding good solutions in a short amount of computation time. A local search has been developed, combining the *Variable Neighborhood Search* meta-heuristic with an *Un-*

feasible Space search.

- Next, a Column Generation method has been developed, to obtain a good lower bound to the best solution value. This step, considered together with the previous one, has helped to tighten the optimal solution and to provide an approximate way to analyze the precision of the first heuristic. To tackle the lengthy procedure of generating columns, a heuristic has been developed for the solution of the Pricing Problem. The method is based on the local search heuristic previously developed, with some further refinements.
- The last part of the development has faced the need of proving the optimality of a solution. A Branch-and-Price approach has been chosen, given the previous two steps. Many technical issues have arisen in bringing this part to completion. The main ones have been found in the procedure needed to induce the models of the local search heuristic and the Column Generation, inherently different, to cooperate with the branching rules.
- Finally a testing phase has been performed, in order to outline the goodness of the choices we have made, both from the theoretical and the implementation point of view. The performance of the algorithms have been proved on several instances and with different variations.

The whole framework has been implemented in COMET. This language has allowed us to write a limited amount of code, compared to the complexity of the system, and to forget about many programming details. However, the language has been developed in an academic environment for research purposes; it is not meant to outperform other more sophisticated or commercial solutions. This aspect, nevertheless, has not been a limit for the conclusions made about our work.

From the experimental tests, improvements in the exact solution of the problem has been shown. Large instances still remain impossible to solve exactly. The VNUS heuristic has performed with high efficiency in all the tests: for all the instances solvable by an exact method the optimum could be found. Of all the LS variations, it has been able to find the best solution on the majority of high dimension instances. It has also revealed a limited sensitivity to changes of the structure of the instances, performing consistently well. The local search implemented for solving the Pricing Problem has proved to possess good scalability with respect to the dimension of the

instance. A bottleneck still remains whenever the Pricing problem is required to be solved to optimality at the last iterations. This makes Column Generation unsuccessful when the dimension of the instances augment. The tests have also revealed how important is to spend efforts in trying to improve the solution of the Pricing Problem. Time needed to solve it is much higher than the one required by the Master Problem, or by the LS heuristic when searching for an upper bound. In the Branch-and-Price the dimension of the instances analyzed in each node tends to becomes smaller as the depth of the tree increases and the quality of the bounds allows to close the tree with a limited number of nodes. However the Branch-and-Price performance depends directly on the two methods used to compute the bounds.

The dimension of the instances that we have been able to solve to optimality has improved with respect to the results obtained in previous works found in the literature.

7.1 Future research guidelines

Further research works can be spent either on the models of the problem or on the algorithms.

A first variation of the problem is obtained by relaxing the partition constraint on the left shore with a cover constraint and by adding a partition constraint on the right shore. We have defined this problem in Chapter 2. Its interpretation can be seen as the opposite of the k-CmBCP: here clients should be grouped together and all the required sessions are sent to all the clients in the group. The reasons for grouping clients can be geographical or technological. A second variation is the weighted generalization of the k-CmBCP, obtained by assigning a weight to the session nodes, or the edges. This new type of problem is very difficult. The weights, in fact, can be used to make the objective function more capable of adhering more realistically to the redundancies introduced by the clustering. One can, for instance, take into account the length of the path (along a multicast tree) from a source node to a client node on which the unnecessary traffic is sent, or the specific capacity or, again, the available bandwidth of each link. It is important to mention that the generalized model with weights on the left shore nodes is exactly the one we have introduced to handle the Branch-and-Price procedure.

About the algorithms, some interesting developments are presented below. The Local Search heuristics can be parallelized easily. From the im-

plementation point of view, Comet offers a `parall` construct to execute some portion of code in parallel. Conceptually there are a few possibilities to obtain a parallel VNS: (i) parallelize the whole procedure; (ii) augment the number of solutions drawn from the current neighborhood and make a local search in parallel from each of them, with or without updating the best solution found; (iii) explore n neighborhoods at the same time and increase the k of n steps each time. The previous expedients can be applied to both the VNUS and the Column Generation heuristic. Another improvement for the Column Generation is the following: let VNUS return more than one solution and use all the columns that can be obtained from the collected solutions to instantiate the RMP. Since solving the RMP does not influence the overall time so much, many starting columns can make the subsequent generation shorter, thus improving performance.

At the end, the delicate choice of the branching variable should be studied further. Choosing λ could reveal a better performance of the procedure, causing less nodes to be generated in the Branch-and-Bound tree. However some technical details have to be addressed, in order, for example, to keep the Pricing Problem linear. Even the LS heuristic should be modified, as forcing a particular cluster to be taken disrupts the neighborhood structures that we have used.

One last note deals with the applications of the k-CmBCP. From the beginning our intention has been to keep an abstract view of the problem, manipulating models in mathematical terms. However a deep knowledge of the application domain could lead to research developments more suited to solve real cases.

Bibliography

[1] Christos H. Papadimitriou, *Computational Complexity.* Addison Wesley, 1982.

[2] Alexandre Guitton, Joanna Moulierac, "Scalable tree aggregation for multicast," in *Proceedings of the 8th International Conference on Telecommunications*, 2005, pp. 129–134.

[3] Jun-Hong Cui, Jinkyu Kim, Dario Maggiorini, Khaled Boussetta, Mario Gerla, "Aggregated Multicast – A Comparative Study," *Cluster Computing*, vol. 8, no. 1, pp. 15–26, February 2005.

[4] Mario Gerla, Aiguo Fei, Jun-Hong Cui, Michalis Faloutsos, *Aggregated Multicast for Scalable QoS Multicast Provisioning.* Springer Berlin / Heidelberg, 2001, vol. 2170/2001, pp. 295–308.

[5] Claude Berge, *Graphs and hypergraphs.* Elsevier Science Ltd, 1985.

[6] Stefano Gualandi, "k-Clustering Minimum Biclique Completion via a Hybrid CP and SDP Approach," in *CPAIOR*, 2009, pp. 87–101.

[7] Nathalie Faure, Philippe Chrétienne, Eric Gourdin, Francis Sourd, "Biclique completion problems for multicast network design," *Discrete Optimization*, vol. 4, no. 3-4, pp. 360–377, December 2007.

[8] James Orlin, *Contentment in graph theory: covering graphs in cliques*, 1977, vol. 80, pp. 406–424.

[9] René Peeters, "The maximum edge biclique problem is NP-complete," *Discrete Applied Mathematics*, pp. 651–654, 2003.

[10] Milind Dawande, Pinar Keskinocak, Sridhar Tayur, "On the Biclique problem in Bipartite graphs," *GSIA*, 1996.

[11] D. Bein, L. Morales, W. Bein, C.O. Shields Jr., Z. Meng, I.H. Sudborough, "Clustering and the Biclique Partition Problem," in *Proceedings of the 41st Annual Hawaii International Conference on System Sciences*. HICSS, 2008, p. 475.

[12] Dimitris Bertsimas, John N. Tsitsiklis, *Introduction to Linear Optimization*. Athena Scientific, 1997.

[13] Jorge Nocedal, Stephen J. Wright, *Numerical optimization*. Springer, 1999.

[14] Mokhtar S. Bazaraa, Hanif D. Sherali, C. M. Shetty, *Nonlinear Programming: Theory and Algorithms*, 3rd ed. Wiley-Interscience, 2006.

[15] Colin R. Reeves, *Modern heuristic techniques for combinatorial problems*. Blackwell, 1993.

[16] Fred Glover, Kochenberger Gary A., *Handbook of Metaheuristics*. Kluwer, 2003.

[17] Christos H. Papadimitriou, Kenneth Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Dover, 1982.

[18] Fred Glover, Manuel Laguna, *Tabu Search*, Springer, Ed. Boston: Kluwer Academic Publishers, 1997.

[19] Pierre Hansen, Nenad Mladenović, José A. Moreno Pérez, "Variable Neighbourhood Search: methods and applications," *4OR*, vol. 6, pp. 319–360, 2008.

[20] Pierre Hansen, Nenad Mladenović, "Variable neighborhood search: Principles and applications," *European Journal of Operational Research*, vol. 130, no. 3, 2001.

[21] Davidon W.C., "Variable metric algorithm for minimization," Argonne National Laboratory, Tech. Rep., 1959.

[22] R. Fletcher and M. J. D. Powell, "A Rapidly Convergent Descent Method for Minimization," *The Computer Journal*, vol. 6, no. 2, pp. 163–168, 1963.

[23] Pierre Hansen, Nenad Mladenović, "First vs. best improvement: An empirical study ," *Discrete Applied Mathematics*, vol. 154, no. 5, pp. 802–817, 2006.

[24] Celso C. Ribeiro, Maurício C. Souza, "Variable neighborhood search for the degree-constrained minimum spanning tree problem," *Discrete Applied Mathematics*, vol. 118, no. 1-2, pp. 43–54, 2002.

[25] Gilles Caporossi, Pierre Hansen, "Variable neighborhood search for extremal graphs: 1 The AutoGraphiX system," *Discrete Mathematics*, vol. 212, no. 1-2, pp. 29–44, 2000.

[26] Pierre Hansen, Nenad Mladenović, Dionisio Perez-Britos, "Variable Neighborhood Decomposition Search ," *Journal of Heuristics*, vol. 7, no. 4, pp. 335–350, 2004.

[27] Alain Hertz, Matthieu Plumettaz, Nicolas Zufferey, "Variable space search for graph coloring," *Discrete Applied Mathematics*, vol. 156, no. 13, pp. 2551–2560, December 2006.

[28] Jacques Desrosiers and Marco E. Lübbecke, *A Primer in Column Generation*. Springer US, 2005, ch. 1, pp. 1–32.

[29] Jinyan Li, Kelvin Sim, Guimei Liu, Limsoon Wong, "Maximal Quasi-Bicliques with Balanced Noise Tolerance: Concepts and Co-clustering Applications," *SIAM*.

[30] Marco E. Lübbecke, Jacques Desrosiers, "Selected Topics in Column Generation," *Operations Research*, vol. 53, no. 6, pp. 1007–1023, 2005.

[31] Laurence A. Wolsey, *Integer programming*. Wiley, 1998.

[32] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh and Pamela H. Vance, "Branch-and-Price: Column Generation for Solving Huge Integer Programs," *Operations Research*, vol. 46, no. 3, pp. 316–329, 1998.

[33] COMET web site, Brown University, http://comet-online.org/.

[34] GroupLens web site, http://www.grouplens.org/.

[35] MovieLens web site, http://movielens.umn.edu/.

[36] CPLEX® web site, ILOG, http://www.ilog.com/products/cplex/.

[37] SCIP web site, ZIB, http://scip.zib.de/.