INAUGURAL-DISSERTATION
zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht-Karls-Universität Heidelberg

vorgelegt von
Ingenieur-Informatik Van Hoai Tran
aus Quang Tri, Vietnam

Tag der mündlichen Prüfung: 2. Juni 2005

Thema

Solving Large Scale Crew Pairing Problems

Gutachter: Prof. Dr. Gerhard Reinelt
Gutachter: Prof. Dr. Dr. h. c. Hans Georg Bock

Abstract

Crew pairing is one of the most critical processes in airline management operations. Taking a timetable as input, the objective of this process is to find an optimal way to partition flights of the timetable without breaking rules and regulations which are enforced by an airline. The problem has attracted many scientists in recent decades. The main challenge is that there is no general method to work well with all kinds of non-linear cost functions and rules.

In order to overcome the non-linearity, the thesis follows a main idea to transfer this combinatorial optimization problem to a set partitioning problem which is one of the most popular $\mathcal{NP}$-hard problems. Although this problem has been studied throughout decades, it becomes more complicated with the increasing size of the input. The complication is induced not only in the transformation process, but also in the methods to solve the resulting set partitioning problem. Finding quickly a good and robust solution for large scale problems is more and more critical to airlines. They are the main targets which are studied by the thesis.

The thesis presents exact methods which are usually based on a branch-and-bound scheme. A branch-and-cut approach applies preprocessing techniques, cutting plane generation methods, and heuristics which are suitable for crew pairing problems. The implementation can solve small and medium sized problems. However, for large problems, a branch-and-price approach is necessary to cope with huge constraint matrices. The thesis improves the weakness of standard column generation methods by applying stabilized column generation variants with sophisticated parameter control schemes into this approach. The computation time is reduced significantly by a factor of three. Moreover, the work also focuses on the extensibility of the methods. This is quite important for large scale problems. Then, we easily obtain a heuristic solution method by controlling running parameters of the presented approaches or combining them together.

Utilizing the available computing resources to deal with large scale crew pairing problems as effective as possible is also a target of the thesis. A new parallel branch-and-bound library is developed to support scientists to solve combinatorial optimization problems. With little effort, they can migrate their sequential codes to run on a parallel computer. The library contains several load balancing methods and control parameters in order to work well with specific problems. The sequential branch-and-cut code to solve set partitioning problems is parallelized by the library and introduces a good speedup for most crew pairing test problems. Parallel computing is also used to solve a so-called pricing subproblem, which is the most difficult problem in the branch-and-price approach, with a nearly linear speedup. The implementation solves large scale crew pairing problems to optimality within minutes, whereas previous methods ended up in the range of hours or more.

Zusammenfassung

Im operativen Betrieb von Fluglinien stellt das Crew-Pairing-Problem eine der grössten Herausforderungen dar. Dieses Problem besteht darin, ausgehend von einem Flugplan, eine optimale Aufteilung der Flüge des Flugplans zu finden, ohne aber Regeln und Vorschriften zu verletzen, die von der Fluggesellschaft eingehalten werden müssen. Crew-Pairing ist Gegenstand intensiver Forschung im Bereich der Mathematik und der mathematischen Informatik. Die besondere Schwierigkeit resultiert aus der hohen Anzahl unterschiedlich formulierter nichtlinearer Kostenfunktionen und Nebenbedingungen. Diese machen die Entwicklung einer generischen, allgemein anwendbaren Methode sehr problematisch.

Ein bekannter Ansatz zur Behandlung der auftretenden Nichtlinearitäten ist die Umformulierung des kombinatorischen Crew-Pairing-Problems in ein Set-Partitioning Problem, welches wiederum eines der am intensivsten studierten $\mathcal{NP}$-schweren Probleme ist. Viele spezielle Strukturen des Set-Partitioning-Problems sind seit Jahrzehnten Forschungsthema und fanden bereits Anwendung bei der Lösung von Crew-Pairing-Problemen. Allerdings steigt die Komplexität des Problems mit der Grösse der Flugpläne stark an. Dies liegt nicht nur an der mathematischen Umformulierung, sondern insbesondere auch an den Methoden, die eingesetzt werden, um die resultierenden Set-Partitioning-Probleme zu lösen. Dabei ist das schnelle und zuverlässige Auffinden von Lösungen für sehr grosse Probleme von zunehmender Bedeutung für die Fluggesellschaften, gerade vor dem Hintergrund der aktuellen Krise in der Luftfahrt und dem daraus resultierenden steigenden Kostendruck. Das schnelle und zuverlässige Auffinden von Lösungen ist Ziel der vorliegenden Arbeit.

Es werden exakte Methoden vorgestellt, die auf dem allgemeinen Branch-and-Bound Schema basieren. Ein hier vorgestellter Branch-and-Cut Ansatz verwendet häufig eingesetzte Preprocessing-Techniken, Cutting-Plane Methoden, sowie spezielle Heuristiken für das Crew-Pairing Problem. Die Implementierung dieses Schemas löst kleine und mittelgrosse Probleme sehr gut. Für sehr grosse Probleme allerdings reicht dieser Ansatz nicht aus. Hierfür wurde ein spezieller Branch-and-Price Ansatz entwickelt, der mit hochdimensionalen Nebenbedingungsmatrizen umgehen kann. Dabei werden in der vorliegenden Arbeit Nachteile von Standardverfahren zur Column-Generation durch den Einsatz von stablilisierten Column-Generation-Verfahren kompensiert, für die eine problemspezifische Parametrierung vorgestellt wird. Die Rechenzeit für betrachtete Probleme konnte mit dieser Technik um den Faktor drei reduziert werden.

Darüberhinaus wurde zusätzlich zu den üblichen Leistungsmerkmalen grosser Wert auf die Erweiterbarkeit der Methoden gelegt. Dies ist ein wichtiger Aspekt in Hinblick auf reale, sehr grosse Probleme. Eine heuristische Lösung spezifischer Crew-Pairing Probleme kann dann relativ leicht durch Anpassen der Programmparameter und durch die Kombination der erwähnten Methoden erhalten werden.

Ein weiteres Ziel dieser Arbeit ist die möglichst effektive Nutzung vorhandener Rechnerressourcen beim Lösen von grossen Crew-Pairing-Problemen. Eine neue Bibliothek von Parallelisierungsroutinen wurde entwickelt, um zukünftig das Bearbeiten

von solchen kombinatorischen Optimierungsaufgaben zu erleichtern und zu beschleunigen. Mit nur geringem Aufwand kann nun ein sequentielles Programm auf einen Parallelrechner migriert werden. Die Bibliothek bietet mehrere Load-Balancing Methoden und Programmparameter an, um die Parallelisierung an das spezifische Problem anzupassen. Die sequentielle Branch-and-Cut Methode liegt ebenfalls als parallelisierte Variante in der Bibliothek vor. Weiterhin wurden Parallelisierungstechniken auf das sogenanntes Pricing-Unterproblem übertragen, dessen Berechnung im übergeordneten Branch-and-Price Verfahren die grössten Schwierigkeiten bereitet. Sämtliche Parallelisierungsansätze zeigen in den betrachteten Beispielen gute Ergebnisse. Mit Hilfe der aktuellen Implementierung kann eine optimale Lösung grosser Crew-Pairing Probleme innerhalb von Minuten gefunden werden, während bisherige Methoden eine Rechenzeit von bis zu Stunden aufweisen.

Contents

List of Tables

List of Figures

List of Algorithms

Introduction

Crew pairing is one of the most critical processes in airline management operations. Taking a timetable as input, the objective of this process is to find an optimal way to partition flights of the timetable without breaking rules and regulations which are enforced by an airline. The difficulties of the problem stay in its inherent properties which have attracted many scientists in recent decades. Dealing with highly non-linear cost functions and rules is always the main topic in this area. The challenge is that there is no general method to work well with all kinds of non-linearity.

In order to overcome the non-linearity, one well-known approach is to transfer this combinatorial optimization problem to a set partitioning problem which is one of the most popular $\mathcal{NP}$-hard problems (Garey and Johnson, 1979). The resulting problem is similar to the process to group elements of a set into subsets governed by additional rules and to maximize the profit from the grouping.

Many special structures of the set partitioning problem have been studied throughout decades and even applied into solving the crew pairing problem. However, the problem becomes more complicated with the increasing size of the input. The complication is induced not only in the transformation process, but also in the methods to solve the resulting set partitioning problem. Although the solution methods are likely to process efficiently small crew pairing problems, they are quite time-consuming with large ones. Moreover, finding quickly a good and robust solution is more and more critical to airlines. They are the main targets which will be studied by the thesis.

Having been investigated for a long period, the crew pairing problem is possibly solved by variety of methods. Generally, they can fall into one of two categories: non-exact or exact methods. Although, from a practical point of view, a better solution is preferred, it is difficult to obtain it in a reasonable execution time and with an available computing resource as well. Therefore, non-exact methods often select a heuristic approach to find a good solution rapidly. By contrast, the thesis will present exact methods which are usually based on the branch-and-bound scheme. The thesis will discuss new advanced algorithms and techniques to obtain the optimal solution. Furthermore, we easily obtain a heuristic solution method by controlling their running parameters or combining them together.

Branch-and-cut and branch-and-price are well-known in the area of combinatorial optimization. In the literature, there are several good results using them to solve the crew pairing problem. Looking back to them, the thesis will present new combinations to create good solution methods. Moreover, along with considering performance as-

pects, the work also focuses on the extensibility of the methods. This is quite important for large scale problems which have been previously mentioned to be of our interest.

Parallel computing is becoming an important part in scientific computing which has emerged as a key technology recently. Although efficient sequential algorithms can provide us with good solutions for crew pairing problems, they cannot utilize much computing resources to deal with large scale crew pairing problems. Solving them in a short time is more demanding, and parallel computing is a good choice in such a case. However, in order to apply it into a specific problem or method, a careful study on sequential methods should be made in advance.

Next, the order and overview of the thesis will be outlined in order to present how the methodological aspects to be applied to the crew pairing problem.

Outline of the thesis

Chapter 1 starts with notations which will be used throughout the thesis. But, the more important substances of the chapter are fundamental theories, algorithms and methods in order to provide useful information to understand other chapters of the thesis. Most of them will be used as the base of the work, and especially, included into implementations.

The crew pairing problem will be described in more details in Chapter 2, including its definition, and main inherent characteristics which make the problem of one airline different from that of another. The literature survey is presented in this chapter in which well-known methodological approaches of the problem will be considered along with their strong and weak points. However, the thesis will emphasize on the exact methods which have been motivated in the introduction. In addition, the chapter will also give a case study, Vietnam Airlines, as a test problem for testing aspects of methodology and implementation of the work.

Branch-and-cut is always one of the most favorite methods to fight a combinatorial optimization problem. An approach presented in Chapter 3 adopts this solution method to solve the crew pairing problem. The implementation in this chapter applies the well-known theory on the facet structure of set partitioning polyhedra. Moreover, other techniques which are widely used in the context of the set partitioning problem are also engaged to accelerate the solution process.

Chapter 4 shows a different approach whose main idea is based on the column generation scheme. Observing the difficulties met by branch-and-cut, branch-and-price only works on a subset of variables, and the remaining will be considered by a so-call pricing subproblem if needed. With this clever idea, the method is expected to work efficiently with larger problems. The chapter details methods for the pricing subproblem which dominates the total computation time. In the context of the crew pairing problem, this subproblem is proven to be $\mathcal{NP}$-hard and is reported (in the literature and this thesis) to take more than 90 percent of the total time. The chapter will pay much attention to this step to find a good method.

Furthermore, Chapter 4 also contributes a new application of stabilized column generation methods to reduce the number of iterations in which control parameters play quite an important role. Empirical experiments on test problems will offer good parameters for the stabilized methods whose implementations in the branch-and-price code and computational results will be presented later in the chapter.

The parallelization of ABACUS, a well-known sequential framework for branch-and-cut-and-price implementations, is described in Chapter 5. The application of parallel computing in a general framework should be more useful to users who have little experience in this area. In order to guarantee the performance, the design of the parallel ABACUS suggests using non-blocking communication as its information exchanging mechanism. The new library is tested with the sequential set partitioning solver in Chapter 3 to solve crew pairing problems.

Chapter 6 dedicates itself to present an effort to speedup the execution time of solving the pricing subproblem. Parallelizing this phase will be shown helpful without worrying the impact of Amdahl's law. The thesis shows the design of a general parallel pricing framework in which we can embed new sequential pricing methods. The implementation aspects are also described in details to obtain good computational performance.

The last chapter closes the thesis with discussions on problems which have been studied in the previous chapters. By this way, readers not only know the study of the author, but also see open problems which should be interesting in future research.

Acknowledgments

The research results presented in this thesis would not have been possible without the support and encouragements of several people. My Ph.D. research was receiving co-operations and helps of many people.

First and foremost, I specially would like to thank Prof. Dr. Dr. h. c. H. G. Bock who has given me a lot of supports throughout the Ph.D. study years. I was quite happy to do research in his innovative research group which has many funny and warm people to keep work moving forward. I am also very grateful Prof. Dr. G. Reinelt for introducing me into combinatorial optimization. He gave me many inspiring and fruitful questions which have helped me much in the research.

With a high respect, I would like to thank Prof. Dr. Sc. H. X. Phu. His support is very critical to my study. Moreover, his life and characters, especially in research, show me useful lessons to live and work as a scientist. I will always remember strong discussions with him which bring me many meaningful things.

I also would like to thank my close friends, T. H. Thai and H. D. Minh, who have shared the study life with me in Germany. Staying a way from the homeland, we are always to be together in study and social life. A thankfulness is also sent to Dr. S. Körkel. He has helped me to contact with the German society through a lot of meaningful favors. The help of P. Kühl to correct the English in the thesis is also highly appreciated.

Working in scientific computing, I highly appreciate those who build and maintain the computing environment. They are T. Kloepfer, M. Neisen, and S. Friedel. The computational results presented in the thesis are always part of their contributions.

I am also grateful to Dr. J. Schlöder, M. Rothfuss, C. Proux-Wieland for helping me to deal with a lot of documents in the university and in life.

And I would like to make a special thank you, and apologies, to any colleagues who I have forgotten.

Finally, I would like to dedicate the work to my parents. They gave me the life, and have taken care of me all their times. Their sacrifices and supports has always asked me to work harder, more effectively and more successfully in order to satisfy them. I am also thankful to all brothers and sisters who devotedly support and encourage me in the whole life.

Chapter 1

Notations and Foundations

This chapter supplies mathematical notations and terminology which will be used to present the approach of the thesis to solve the crew pairing problem. Theoretical foundations of necessary areas are also included here with emphasis on definitions and propositions which will be the base for further considerations. Since the chapter will certainly not cover all things needed, additional references will be given throughout the thesis.

1.1 Linear Algebra

The *real number line* is denoted by $\mathbb{R}$ and the *n-dimensional real vector space*, the set of ordered n-tuples of real numbers, by $\mathbb{R}^n$. Such a tuple is called a *vector* of size n and locates a point in $\mathbb{R}^n$. The i-th component of a vector x is denoted by x_i. We use the notation $x^\top y$ to present the inner product of two vectors x and y.

An $m \times n$ matrix A with element A_{ij} in Row i and Column j is written $A = [A_{ij}]$. We denote by $A_{i\cdot}$ the ith row vector of the matrix, $A_{\cdot j}$ the jth column vector of the matrix. Throughout the thesis, we often assume that $A_{ij} \in \mathbb{R}$. The *transpose* of the matrix is written as $A^\top$.

A set of points $x^1, \ldots, x^k \in \mathbb{R}^n$ is *linearly independent* if the unique solution of $\sum_{i=1}^k \lambda_i x^i = 0$ is $\lambda_i = 0,\ i = 1, \ldots, k$. Note that the maximum number of linearly independent points in $\mathbb{R}^n$ is n. The maximum number of linear independent rows (columns) of a matrix A is the *rank* of A, and is denoted by $\text{rank}(A)$.

A set of points $x^1, \ldots, x^k \in \mathbb{R}^n$ is *affinely independent* if the unique solution of $\sum_{i=1}^k \alpha_i x^i = 0, \sum_{i=1}^k \alpha_i = 0$ is $\alpha_i = 0, i = 1, \ldots, k$. Note that the maximum number of affinely independent points in $\mathbb{R}^n$ is $n + 1$.

A *(linear) subspace* is a subset of a vector space R^n which is closed under vector addition and scalar multiplication. An *affine subspace* A is a linear subspace S translated by a vector u, that is, $A = \{u + x : x \in S\}$. The dimension $\dim(S)$ of a subspace S is equal to the maximum number of linearly independent vectors in it. An affine subspace A translated from a linear space S also has a dimension, $\dim(A)$, which is $\dim(S)$.

A proposition describing the relationship between a subspace and linear systems is that the statements below are equivalent:

(i) $H \subseteq \mathbb{R}^n$ is a subspace.

(ii) There is an $m \times n$ matrix A such that $H = \{x \in \mathbb{R}^n : Ax = 0\}$.

(iii) There is a $k \times n$ matrix B such that $H = \{x \in R^n : x = uB, u \in \mathbb{R}^k\}$.

1.2 Graph Theory

A *graph* $G = (V, E)$ consists of a finite, nonempty set $V = \{v_1, \ldots, v_m\}$ and a set $E = \{e_1, \ldots, e_n\}$ whose elements are subsets of V of size 2, that is $e_k = (v_i, v_j)$, where $v_i, v_j \in V$. The elements of V are called *nodes* (*vertices*), and the elements of E are called *edges* (*links*).

A *directed graph* or *digraph* $D = (V, A)$ consists of a finite, nonempty set $V = \{v_1, \ldots, v_m\}$ and a set $A = \{e_1, \ldots, e_n\}$ whose elements are *ordered* subsets of V of size 2 *arcs*.

Under the definition of a graph $G = (V, E)$, several terms for its components should be given. If $e = (v_i, v_j) \in E$, we say v_i is *adjacent* to v_j (and vice-versa) and that e is *incident* to v_i (and v_j). For an undirected graph, $\Gamma(v)$ denotes the set of incident vectices of v. For a digraph, notations $\Gamma^{-1}(v)$ and $\Gamma^{+1}(v)$ are used for the set of incoming vertices and outgoing vertices respectively. The *degree* of a vertex v of G is the number of edges incident to v, also say $|\Gamma(v)|$. We denote by $N(v)$ the set of vertices adjacent to v.

Subsets of nodes and edges are also important so that we need some definitions for them. A *walk* in a graph $G = (V, E)$ is a sequence of nodes $v_1 \ldots v_k, k \geq 1$, such that $(v_i, v_{i+1}) \in E, i = 1, \ldots, k-1$. A walk without any repeated nodes is called a *path*. A *circle* is a closed walk which has no repeated nodes other than its first and last one. Similarly, one can define corresponding terms for direct graphs.

There are many kinds of graph in theory and practice. In this preliminary section, we just show some basic kinds of graph:

(i) A graph is *connected* if there is a path between each pair of its nodes.

(ii) A graph is said to be *acyclic* if it does not contain any cycles. The graph is also called *forest*.

(iii) A connected forest is called a *tree*.

We usually work with *weighted* graphs which are associated with a function $w : E \longrightarrow \mathbb{R}$ (usually just $\mathbb{R}^+ = \{x \in \mathbb{R} : x \geq 0\}$).

Reference to the graph theory can be found in many textbooks in computer science and mathematics.

1.3 Convex Programming

Firstly, a general definition of an *optimization problem* is given as following:

Definition 1.3.1. An instance of an optimization problem is a pair (F, c), where F is the set of feasible points, c is the cost function, a mapping $c : F \longrightarrow \mathbb{R}$. The problem is to find an $f \in F$ for which

$$c(f) \leq c(y) \ \forall y \in F.$$

Such a point is called a *globally optimal solution* to the given instance.

Note that the words "*programming*" and "*optimization*" have the same meaning in applied mathematics. Restricting the feasible region and the cost function gives different classes of optimization problem. In order to describe convex programming, we need two definitions below.

Given two points $x, y \in \mathbb{R}^n$, a *convex combination* of them is any point of the form

$$z = \lambda x + (1 - \lambda)y, \qquad \lambda \in \mathbb{R} \text{ and } 0 \leq \lambda \leq 1.$$

If $\lambda \neq 0, 1$, we say z is a *strict* convex combination of x and y. A set $S \subseteq \mathbb{R}^n$ is *convex* if it contains all convex combinations of pairs of points $x, y \in S$. The *convex hull* of a set S, denoted by $\text{conv}(S)$, is the set of all points which are convex combinations of points in S.

Let $S \subseteq \mathbb{R}^n$ be a convex set. The function $c : S \longrightarrow \mathbb{R}$ is *convex in* S if for any two points $x, y \in S$

$$c(\lambda x + (1 - \lambda)y) \leq \lambda c(x) + (1 - \lambda)c(y), \qquad \lambda \in \mathbb{R} \text{ and } 0 \leq \lambda \leq 1.$$

A function c defined in a convex set $S \subseteq \mathbb{R}^n$ is called *concave* if $-c$ is convex in S.

Convex programming regards the minimization of a convex function on a convex set. If the convex feasible region is defined by a set of inequalities involving concave functions, the optimization problem is called a convex programming problem. More precisely, the definition of convex programming is as follows:

Definition 1.3.2. An instance of an optimization problem (F, c) is a convex programming problem if c is convex and $F \subseteq \mathbb{R}^n$ is defined by

$$\begin{aligned} h_i(x) &= 0 \quad i \in \mathcal{H} \\ g_j(x) &\geq 0 \quad j \in \mathcal{G} \end{aligned}$$

where $h_i : \mathbb{R}^n \longrightarrow \mathbb{R}$ are linear functions, and $g_j : \mathbb{R}^n \longrightarrow \mathbb{R}$ are concave functions. Note that $\mathcal{H}, \mathcal{G}$ are sets of indices.

These problems have the interesting property that local optima are global. More precisely, we describe an associated proposition below:

Proposition 1.3.3. *In a convex programming problem, every point locally optimal with respect to the Euclidian distance neighborhood N_ϵ which is defined by*

$$N_\epsilon(x) = \{y : y \in F, ||x - y|| \leq \epsilon\}$$

is also globally optimal. (Note that generally a neighborhood is a mapping $N : F \longrightarrow 2^N$.)

We also have conditions for optimality that are sufficient, the Kuhn-Tucker conditions.

Further discussion of convex analysis can be found in Hiriart-Urruty and Lemaréchal (1993). The book covers not only the aspects of property, but also practical algorithms. Convex programming is presented in some books about the combinatorial optimization, such as Papadimitriou and Steiglitz (1998), Nocedal and Wright (1999), Nemhauser and Wolsey (1988).

1.4 Polyhedral Theory

The theory focuses on investigating the feasible set of optimization problems which are described by a set of linear inequalities.

Definition 1.4.1. A *polyhedron* $P \in \mathbb{R}^n$ is the set of points which satisfy a finite number of linear inequalities, that is, $P = \{x \in \mathbb{R}^n : Ax \leq b\}$, where (A, b) is an $m \times (n + 1)$ matrix. A *polytope* is a *bounded* polyhedron (i.e., $\exists \omega \in \mathbb{R}^+ : P \subseteq \{x \in \mathbb{R}^n : -\omega \leq x_j \leq \omega, j = 1, \ldots, n\}$).

It is quite clear that a polyhedron is a convex set. A polyhedron $P \in \mathbb{R}^n$ is called *full dimensional* if $\dim(P) = n$. Note that $\dim(P)$ is the dimension of the smallest affine space that contains the polyhedron.

Consider a polyhedron $P = \{x \in \mathbb{R}^n : Ax \leq b\}$. In order to find out necessary inequalities to describe the polyhedron, we should address some definitions for inequalities.

Definition 1.4.2. The inequality $\pi x \leq \pi_0$ (abbr. (π, π_0)) is called *valid inequality* for P if it is satisfied by all $x \in P$. $\{x \in \mathbb{R}^n : \pi x = \pi_0\}$ is called a *hyperplane*. A hyperplane divides $\mathbb{R}^n$ into two *hyperspaces*:

$$\{x \in \mathbb{R}^n : \pi x \leq \pi_0\}$$
$$\{x \in \mathbb{R}^n : \pi x \geq \pi_0\}.$$

Definition 1.4.3. If $\pi x \leq \pi_0$ is a valid inequality for P, and $F = \{x \in P : \pi x = \pi_0\}$, F is called a *face* of P, and we say that $\pi x \leq \pi_0$ *represents* F.

We have three different kinds of faces:

- A *facet* is a face of dimension $n - 1$.

- A *vertex* is a face of dimension zero (a point).
- An *edge* is a face of dimension one (a line segment).

Foundation on the polyhedral theory can be found in Nemhauser and Wolsey (1988). The book is quite interesting to review ways of describing polyhedra by facets.

1.5 The Simplex Method

The class of the convex programming problem mentioned in Section 1.3 covers a problem to be discussed in this section, the *linear programming problem*. The feasible convex region of a linear program is defined by a polyhedron and the cost function is linear as following:

$$\text{(LP)} \qquad \begin{array}{rll} z^* = & \min & c^\top x \\ & \text{s.t.} & Ax = b \\ & & x \geq 0, \end{array} \tag{1.1}$$

where A is a $m \times n$ matrix of elements $A_{ij} \in \mathbb{Q}$, and $c, b \in \mathbb{Q}^n$. Note that m, n are positive integers. Problem (1.1) is said to be in the *standard form* (i.e., there are only equalities). It is quite simple to prove that the *general form* of a linear program in which both equalities and inequalities exist is equivalent to the standard form of that problem.

Linear programming plays an important role in optimization. Many real applications can be modelled as linear programs or *integer programs* which are usually solved by relaxation to linear programs. There are two main algorithms for solving LPs: the simplex method and interior point methods. In this thesis, we only focus on the simplex method and its aspects. Two good text books for more details on interior point methods are of Vanderbei (2001) and Nocedal and Wright (1999).

The simplex method was invented by Dantzig in 1949 and is described in his book (Dantzig, 1963). In spite of having an exponential complexity (Klee and Minty, 1972), it has been revisited by scientists and been improved much to apply to practical applications throughout six decades. Therefore, there are variants of the simplex method which cannot all be included in the thesis. Instead, only a basic knowledge of the method will be reviewed.

Before going further to the simplex method, we need several definitions.

Definition 1.5.1. Without loss of generality, we can assume A is of rank m.

(i) A $m \times m$ nonsingular matrix $A_B = (A_{.B_1}, \ldots, A_{.B_m})$ is called a *basis*, where $B = \{B_1, \ldots, B_m\}$ is the set of column indices. Let $N = \{1, \ldots, n\} \setminus B$. Now permute the columns of A, we can write $Ax = b$ as $A_B x_B + A_N x_N = b$, where $x = (x_B, x_N)$.

(ii) The solution $x_B = A_B^{-1} b$, $x_N = 0$ is called a *basic solution* of $Ax = b$.

(iii) x_B is the vector of *basic variables* and x_N is the vector of *nonbasic variables*.

(iv) If $A_B^{-1}b \geq 0$, then (x_B, x_N) is called a *basic primal feasible solution* and A_B is called a *primal feasible basis*.

(v) Let $\bar{A}_N = A_B^{-1}A_N$. Then $\bar{c}_N = c_N - c_B^\top \bar{A}_N$ is called the *reduced cost* vector for nonbasic variables.

In order to find an optimal solution, the main idea of the simplex method is to *pivot* from one basis to another adjacent basis which is defined to be different in only one column with the previous basis. An outline of the algorithm can be as Algorithm 1.5.1. In Step 1, we possibly have to introduce artificial variables to obtain the first feasible basis. The idea is to create another linear program which is feasible. The result of the new problem will be used as the starting point for the first problem. Another question is how to choose which column r enters the basis. One of approaches is to select the most negative $\bar{c}_r$ which corresponds to a kind of *steepest descent* policy.

Algorithm 1.5.1 The (primal) simplex method

1: Start with a primal feasible basis A_B. {*Initialization*}
2: **while** $\bar{c}_N < 0$ **do** {*Optimality testing*}
3: Choose an $r \in N$ with $\bar{c}_r < 0$. {*Pricing*}
4: **if** $\bar{A}_{\cdot r} \leq 0$ **then** {*Unboundedness Testing*}
5: $z^* = -\infty$.
6: **break**
7: **else** {*Basis pivoting*}
8: Find the unique adjacent primal feasible basis $A_{B^{(r)}}$ that contains $A_{\cdot r}$.
9: Let $B \leftarrow B^{(r)}$.
10: **end if**
11: **end while**

Textbooks of Vanderbei (2001), Nocedal and Wright (1999) are preferred for much more details in the classical simplex method and its variants. Short and clear introduction to linear programming is also given by Nemhauser and Wolsey (1988), Papadimitriou and Steiglitz (1998). Their books are quite useful if one wants to consider linear programming within the context of integer programming.

1.6 Integer Programming

Variables of optimization problems fall into two categories: *continuous* variables, and *discrete* variables. We call those problems with discrete variables *combinatorial optimization problems* which look for an object from a *finite*, or possibly countably infinite, set. Loosely speaking, dealing with different kinds of variables of optimization problems requires different methods to solve them "efficiently".

In Section 1.5, we have defined a linear programming problem and mentioned its close relationship to the following optimization problem:

Definition 1.6.1. An optimization problem written as

$$\text{(MIP)} \qquad \begin{array}{ll} \min & c^\top x + h^\top y \\ \text{s.t.} & Ax + Gy = b \\ & x \in \mathbb{Z}^n_+,\ y \in \mathbb{R}^p_+ \end{array} \tag{1.2}$$

is called a *linear mixed integer program*. If there is no continuous variable, the problem is called a *linear integer programming problem*, or integer program (IP) for short. If, in an (IP), we restrict the variables x as $x \in \{0,1\}^n$, we have a *binary problem*.

Solving an IP involves not only general algorithms but also problem specific techniques. General algorithms can be classified into 2 types: *non-exact* algorithms and *exact* ones. Non-exact methods are mainly based on *heuristics* or approximation algorithms. Greedy and local search algorithms could be the first choice of researchers to attack a combinatorial optimization problem. Unfortunately, under the local search mechanism, we can get trapped in a local minimum. In order to escape from that, we need an improved heuristic. Some techniques which help us in such a case are: *tabu search*, *simulated annealing*, *generic algorithms*. They are called *meta-heuristics*.

Tabu search was started by Glover (1986) and strongly discussed in two following texts (Glover, 1989, 1990). The idea behind the method is to avoid cycling in the process of moving among feasible solutions. A basic version of the tabu search algorithm can be in Algorithm 1.6.1.

Algorithm 1.6.1 A tabu search algorithm

1: Initialize an empty tabu list. Get an initial solution x.
2: **while** (stopping condition not reached) **do**
3: Choose a subset $N'(x) \in N(x)$ of non-tabu solutions.
4: Let $x' = \arg\min\{c(y) : y \in N(x)\}$.
5: $x \leftarrow x'$ and update the tabu list.
6: **end while**

Note that $N(x)$ in the algorithm denotes a predefined neighborhood of x and it is usually problem dependent.

Simulated annealing approaches in a different way by choosing randomly a neighbor to move to. Precisely, the neighbor will be chosen with the probability of 1 if it has a better cost value, otherwise, with some probability strictly between 0 and 1. An outline of the simulated annealing method can be found in Algorithm 1.6.2. The method is described in Metropolis et al. (1953), Kirkpatrick et al. (1983).

Genetic algorithms originated with the work of Holland (1975), Goldberg (1989). Not working with an individual solution, at each iteration, the method considers a finite set of solutions (called *population*) and this set will change randomly from one generation to the next. A sketch of the method can look like Algorithm 1.6.3.

If one prefers an exact approach, *cutting plane algorithms* and *branch and bound algorithms* are widely used methods. Furthermore, they can also be used in non-exact

Algorithm 1.6.2 A simulated annealing algorithm

Initialize a temperature T and a reduction factor $r \in (0, 1)$.
Get an initial solution x.
while (not yet frozen) **do**
  Pick a random neighbor $x' \in N(x)$. Let $\Delta = c(x') - c(x)$.
  if $\Delta \leq 0$ **then**
    $x \leftarrow x'$
  else
    $x \leftarrow x'$ with probability $e^{-\Delta/T}$.
  end if
  Reduce the temperature $T \leftarrow rT$.
end while

heuristics. Given an (IP): $\min\{c^\top x : Ax = b\}$, a cutting plane method tries to find out the description of the convex hull of the set of feasible solutions. The method iteratively cuts off fractional points of the polyhedron $P = \{x \in \mathbb{R}^n : Ax = b\}$ by valid inequalities. In order to understand the algorithm, we must perceive the following definition:

Definition 1.6.2. The *separation problem* associated with a combinatorial optimization problem $\min\{c^\top x : x \in X \subseteq \mathbb{R}^n\}$ is the problem: given $x^* \in \mathbb{R}^n$, is $x^* \in \text{conv}(X)$? If not, find a valid inequality $\pi^\top x \leq \pi_0$ for X, but violated by the point x^*.

The equivalence of optimization and separation is shown by Grötschel et al. (1981).

In Algorithm 1.6.4 which describes a basic cutting plane algorithm, $\mathcal{F}$ in the algorithm denotes a family of valid inequalities for X. Certainly, the algorithm can terminate without finding an integral solution. Moreover, although several cutting plane algorithms (e.g., the cutting plane algorithm with Gomory cuts) were proved, under some circumstances, to be of finite convergence, it is still not practical to continue the loop until no violated valid inequality is found. The exit from the loop gives

Algorithm 1.6.3 A genetic algorithm

Initialize a population $X = \{x^1, \ldots, x^k\}$.
while (stopping condition not reached) **do**
  (Evaluation): evaluate the *fitness* of the individuals of X.
  (Parent selection): select certain pairs of solutions (called *parents*) $P' \subseteq P = \{(x^i, x^j) : x^i, x^j \in P\}$ based on their fitness.
  (Crossover): each pair combines to create one or two new solutions.
  (Mutation): modify randomly some of the new solutions.
  (Population selection): based on their fitness, create a new population X' by replacing some or all individuals of the old population X.
end while

an improved formulation which can be the input of a branch and bound method to be discussed in the next section.

Algorithm 1.6.4 A cutting plane algorithm

1: $t \leftarrow 0$, $P^0 \leftarrow P$.
2: **while** (stopping condition not reached) **do**
3: Solve the linear program $x^t = \arg\min_x\{c^\top x : x \in P^t\}$
4: **if** $x^t \in \mathbb{Z}^n$ **then**
5: x^t is an optimal solution.
6: **break**
7: **else**
8: Solve the *separation problem* for x^t and the family $\mathcal{F}$.
9: **if** $\exists(\pi^t, \pi_0^t) \in \mathcal{F}$ such that ${\pi^t}^\top x > \pi_0^t$ **then**
10: $P^{t+1} \leftarrow P^t \cap \{{\pi^t}^\top x \leq \pi_0^t\}$.
11: $t \leftarrow t + 1$.
12: **else**
13: **break**
14: **end if**
15: **end if**
16: **end while**

Links to integer programming can be found in many textbooks on combinatorial optimization or integer programming, such as Nemhauser and Wolsey (1988), Papadimitriou and Steiglitz (1998), Wolsey (1998). A book of Kallrath and Wilson (1997) which shows the integer programming problem under the application's point of view could be helpful to those who want to solve practical problems.

1.7 Branch-and-Bound

The so-called *branch-and-bound* method is based on two following conclusions:

Proposition 1.7.1. *Consider the problem* $z = \min_x\{c^\top x : x \in X\}$. *Let* $X = X_1 \cup \ldots \cup X_K$ *be a decomposition of* X *into smaller sets, and let* $z^k = \min_x\{c^\top x : x \in X_k\}$ *for* $k = 1, \ldots, K$. *Then* $z = \min_k z^k$.

Proposition 1.7.2. *Use the same notations as in Proposition 1.7.1. Let* $\underline{z}^k$ *be a lower bound on* z^k, *and* zo^k *be an upper bound on* z^k. *Then* $zu = \min_k \underline{z}^k$ *is a lower bound on* z *and* $\overline{z} = \min_k \overline{z}^k$ *is an upper bound on* z.

With the help of Proposition 1.7.2, many smaller problems finding z^k do not need to be solved or be stopped quickly because they violate some bounds. The idea is called *implicit enumeration* which contrasts to *explicit enumeration* which explores totally the feasible region. It is not necessary to solve smaller problems to optimality. Instead,

upper or lower bounds are enough for the branch-and-bound method (otherwise, we can decompose those problems to smaller parts). Hence, a *relaxation* of an integer problem (IP) is needed, being defined as an optimization problem (RP) with the feasible set $X_R \supseteq X$, and the cost function $z_R(x) \leq c^\top x$ for $x \in X$. Algorithm 1.7.1 gives a general branch-and-bound algorithm for solving (IP): $\min_x\{c^\top x : x \in X\}$.

Algorithm 1.7.1 A general branch-and-bound algorithm

1: $\mathcal{L} \leftarrow \{(\text{IP})\}$, $X^0 \leftarrow X$, $\underline{z}^0 \leftarrow -\infty$, $\overline{z}_{\text{IP}} \leftarrow \infty$.
2: **while** $(\mathcal{L} \neq \emptyset)$ **do**
3: Select and delete a problem $(\text{IP})^{\text{i}}$ from $\mathcal{L}$. {*Problem selection*}
4: Solve its relaxation $(\text{RP})^{\text{i}}$ with optimal value z_R^i, and optimal solution x_R^i (if they exist). {*Problem relaxation*}
5: **if** $(z_R^i < \overline{z}_{\text{IP}})$ **then** {*Bounding*}
6: **if** $(x_R^i \in X^i$ and $c^\top x_R^i < \overline{z}_{\text{IP}})$ **then**
7: $\overline{z}_{\text{IP}} \leftarrow c^\top x_R^i$.
8: Delete from $\mathcal{L}$ all problems with $\underline{z}^i \leq \overline{z}_{\text{IP}}$. {*Fathoming*}
9: **if** $(c^\top x_R^i = z_R^i)$ **then**
10: **continue**
11: **end if**
12: **end if**
13: Decompose X^i into $\{X^{ij}\}_{j=1}^k$ and add associated problems $\{(\text{IP})^{\text{ij}}\}_{\text{j=1}}^{\text{k}}$ to $\mathcal{L}$, where $\underline{z}^{ij} = z_R^i$ for $j = 1, \ldots, k$. {*Branching*}
14: **end if**
15: **end while**

In the algorithm, there are many rising questions, such as how to select the next problem $(\text{IP})^{\text{i}}$, how to decompose X^i. Choosing a "good" relaxation is also a concern. Linear relaxation is often a choice in many branch-and-bound codes partly due to the linear presentation of the remaining problem if we drop the integral constraints of an integer problem.

Obviously, exploring a branch-and-bound tree of an instance of an optimization problem is a time consuming task, especially with $\mathcal{NP}$-hard problems. Not only being discussed in many textbooks on integer programming, branch-and-bound is also implemented in many application codes. Developers often prefer to build branch-and-bound as a general framework (Nemhauser et al., 1994, Thienel, 1995, Ralphs, 2001) from which users can embed effortlessly their own application codes.

Parallel computing described in the next section is one of many methods to utilize computing resources of parallel machines to explore branch-and-bound trees.

1.8 Parallel Computing

Loosely speaking, *parallel computing* is concerned with using groups of computing elements simultaneously. Unlike sequential computing, involving many processors into

computation raises the issue of how all processors cooperate efficiently to solve application problems. Several aspects of parallel computing which should be considered are: parallel architectures, parallel algorithms, parallel programming languages and performance analysis.

Since the thesis pays attention to an application in airline management, it will not show much details in parallel architecture. Readers can easily find many good books on this area, for example Hwang (1984), Hwang and Xu (1998). The level of algorithms can be said one of the most important ingredients for parallel computing. A parallel algorithm describes how to partition a given problem into smaller problems, how to communicate among processors, and how to join the partial solutions to produce the final result. We should agree with several basic definitions for further understanding:

Definition 1.8.1.

(i) A *process* is a sequence of program instructions performed in sequence within an operating system.

(ii) A *processor* is a hardware element to execute program instructions

Sometimes, we use these two terms synonymously. This happens when only one process is mapped onto a processor. In such a case, almost all computing power of the processor is dedicated to the process (there are other light control processes of the operating system).

Considering the *precedence graph* of a computation is quite essential for parallel computing. The graph views the dependencies of computation blocks in which the order of blocks must be satisfied to produce the result correctly.

In a multiprocessor environment, the way of processors communicating with each other has two possibilities:

(i) *Message passing* communication: processors communicate via communication links.

(ii) *Shared memory* communication: processors communicate via common memory.

Evolution of *cluster computing* has given the message passing model an advantage over shared memory counterpart due to its extendibility and flexibility. Most of parallel computers built in recent years are off-the-shelf clusters. In order to develop parallel algorithms on these systems, users can choose one of several message passing communication libraries, such as PVM (Geist et al., 1994), or MPI (Message Passing Interface Forum, 1995).

One of the remaining important issues is performance analysis which observes computational results in order to improve the efficiency or to measure the quality of parallel algorithms.

Definition 1.8.2. A *speedup* of a parallel computation utilizing p processors is derived as the following ratio:

$$S_p = \frac{t_s}{t_p}$$

where t_s is the execution time to perform the computation on one processor and t_p is the execution time needed to do the same computation using p processors. (Sometimes, t_s is defined as the time spent by the "best" sequential algorithm.)

The efficiency of a parallel computation is defined as follows:

$$E_p = \frac{S_p}{p}$$

A "good" parallel computation should deliver a speedup near to the number of processors in use. Normally $1 \leq S_p \leq p$, however, in practice, a speedup greater than p processors can be observed and is called a *super-linear speedup*.

An excellent online book is presented by Foster (1995). Other helpful textbooks are Hwang and Xu (1998), Roosta (1999). The standards published by Message Passing Interface Forum are quite effective for those who implement parallel codes on cluster systems.

Chapter 2

The Crew Pairing Problem

2.1 Airline Management Operations

The efficient management of airline operations has become more challenging and complex than ever before, particularly in today's dynamic and often unpredictable air travel industry. Although mathematical programming tools have been applied in this area for several decades, its problems are still challenging scientists and software engineers. The size of these problems is increasing and restrictions on them are becoming more and more complicated. On the other side, airlines always want to solve them as well as possible in order to reduce the operating cost and raise the revenue. Moreover, the airline industry has become more time-demanding. Solutions for its management operations have to be available readily. These challenges have motivated the research presented in this thesis.

We suppose that a timetable of flights operated in a schedule period exists already to match the expectations of the market demands. Then, there are planning and scheduling tasks for aircraft and crews. The first problem is called *fleet assignment problem* and has the timetable as input. The departure times are not necessary to be fixed. Time windows are often used for them to give a more robust solution. Given a homogenous aircraft fleet (a group of aircraft of the same type), the aim of the fleet assignment step is to find an aircraft schedule to maximize the profits which are characterized by a costs/revenues function to map aircraft to flight legs. The assignment of individual aircraft has to satisfy additional constraints. For example, an airline company has a limited number of aircraft for a certain type, certain aircraft are not allowed to operate at certain times and airports, a minimum ground time of each aircraft type at a certain airport has to be respected, etc. The results of the fleet assignment problem are: the exact departure time for each flight leg, the sequence of flight legs for an aircraft. A general review on the fleet assignment problem is found in Rushmeier et al. (1995). An application of branch-and-bound with column generation method for aircraft routing and scheduling is shown by Desaulniers et al. (1997b). Another work on the problem was reported at Delta Airlines by Subramanian et al. (1994).

Crew cost is the second largest portion of the overall-cost, next to fuel cost. Practical reports say that major airlines in the United States and Europe spend about $1 billion for crew cost. Therefore, the second problem called *crew scheduling problem* is very important. This problem is often divided into 2 smaller problems: *crew pairing problem* and *crew rostering problem* (also called *crew assignment problem*). The crew pairing problem takes the scheduled flights which were fixed by the fleet assignment step as input. Instead of assigning aircraft, the aim now is to allocate crews to cover all flight legs and maximize an objective function. The feasible region of the solution search is limited by a complicated set of rules coming from several sources. In the crew pairing process, planners do not consider individual crew and the scheduling is often applied for a period. The result of this process will be used for different periods within a session. The flights are grouped into small sets called *pairings* (also called *rotations*) which must start from a home base and end at that base. A pairing often has few flights legs (usually ≤ 6). The rostering process will do the remaining task to assign an individual crew to a flight leg. Since it is the final step, all constraints must be involved in. In reality, the problem is interpreted as assigning the generated pairings to named individuals considering all their activities such as training requirements, vacations, etc. The objective of this step is often to maximize the utilization of crew (to reduce the number of crew involved). Remember that the rostering problem must take care of any changes between daily or weekly schedules to create a fully dated schedule.

An example of a crew pairing problem

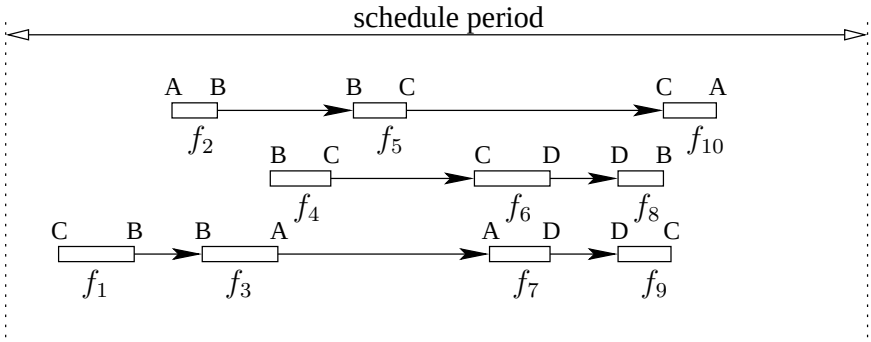


Figure 2.1: An example timetable for a crew pairing problem

Let us consider the timetable in Figure 2.1. There are 3 aircraft routes that possibly result from a previous fleet assignment problem. The problem is to find a "good" set of pairings to cover all the flights in the timetable assumed to be operated in a given time period (e.g., a day, a week). A simple set of airline rules can be as following:

- Rule 1: a pairing cannot have more than 4 legs,
- Rule 2: a pairing must last less than the given schedule period,

- Rule 3: the ground time between two consecutive flights must not be less than 40 minutes.

Assume that the set of home base airports is {A,B}. Under those assumed rules, we can easily see the set P of feasible pairings for the timetable:

$p_1: \ f_2 \to f_5 \to f_{10}$	$p_8: \ f_4 \to f_1$
$p_2: \ f_2 \to f_4 \to f_{10}$	$p_9: \ f_5 \to f_{10} \to f_2$
$p_3: \ f_3 \to f_7 \to f_9 \to f_1$	$p_{10}: \ f_5 \to f_6 \to f_8$
$p_4: \ f_3 \to f_7 \to f_8$	$p_{11}: \ f_5 \to f_6 \to f_9 \to f_1$
$p_5: \ f_4 \to f_6 \to f_8$	$p_{12}: \ f_5 \to f_1$
$p_6: \ f_4 \to f_6 \to f_9 \to f_1$	$p_{13}: \ f_7 \to f_9 \to f_1 \to f_3$
$p_7: \ f_4 \to f_{10} \to f_2$	$p_{14}: \ f_7 \to f_8 \to f_3$

Two of solutions for the crew pairing problem are $\{p_1, p_3, p_5\}$ and $\{p_1, p_4, p_6\}$. With a given cost function, we can choose the best among all solutions if it is feasible (it is the case in the example). In the list of pairings, you cannot see the pairing $f_2 \to f_4 \to f_6 \to f_8 \to f_3$ though it starts from base A and ends at it. The pairing violates rule 2 in the rule set. The pairings p_8, p_9 would be invalid in case that the ground time between f_5 and f_6 is less than 40 minutes mentioned in rule 3. We can see that certain pairings (e.g., p_5, p_9) pass the time period boundary in order to connect flights. This is easily understood because we assume the timetable is the same in every period, which is also the assumption of other crew pairing problems.

In the next section, we will review characteristics which characterize up a crew pairing problem. Before going to the part of finding solution methods, the understanding of the problem will help us to solve it effectively.

2.2 Characteristics of the Crew Pairing Problem

As defined in the previous section, the main task of crew pairing can be thought as finding a set of subsets of a ground set to "cover" the ground set with minimal cost. The cost of a subset is given according to a specific problem. In a strict version, all the subsets must be disjoint as in a set partitioning problem. In a crew pairing problem, the ground set corresponds to the set of flight legs to be scheduled in a time period, and the subsets are pairings. Moreover, the crew pairing problem does not allow every subset to be used as a candidate. It requires that pairings must conform to safety rules, contractual agreements and regulations of a company. Another difference to standard set partitioning problems is that the ground set in a crew pairing problem is an ordered set according to the departure time of flights.

We will see main factors which shape a crew pairing problem:

- Crew category and fleet (aircraft types): a crew often consists of different categories, for example, cockpit crew and cabin crew. Different crew categories are

obviously restricted by different airline rules and paid by different salaries. It is reasonable to consider different categories as different problems. The problem for cabin crew is often larger than that of cockpit crew, but the latter costs more than the former does. Moreover, in all airlines, a crew is often qualified to fly a few number of aircraft types due to some reasons of safety, effectiveness and low cost in airline management operations. Cabin crews are allowed to serve on more fleets than cockpit crews do. Therefore, the crew pairing problem is often decomposed by crew category and fleet.

- Regularity of timetable: this property also determines the size of a crew pairing problem. Many papers (e.g., Anbil et al. (1991a), Andersson et al. (1997)) say that major U.S. airlines operate daily timetables with a large number of flight legs. European airlines often provide weekly regular timetables. Vietnam Airlines, as being seen later, operates a weekly timetable. Although there are few differences between these two kinds of regularity, they can be considered to repeat in every given time period (e.g., day, week). After creating pairings for that period, we can use them for the rostering process which uses many other factors as well in order to create a real schedule. Certainly, it is quite unreasonable to build a weekly problem by putting a sequence of 7 same daily timetables together. The result of the problem could be improved (according to a cost structure), but will be quite time-consuming.

- Network structure: the operational flight network is decided by market demands and capabilities of airlines. However, it also affects the crew pairing step. Most airlines follow the *hub-and-spoke* idea. A connection between two non-hub airports must go through a hub. Besides economic reasons, this design helps to deal with disruptions which often happen nowadays. For example, if there is a pool of crew resources at a hub, airlines can overcome flight delays in some extents. Since a star-shaped flight network presents more possibilities for outgoing connections, there will be more pairings for its optimization problem. This will be seen in more details in the solution chapters.

- Rules and regulations: any airlines company has its own set of rules to define how a pairing is feasible. Rules can come from governmental obligations, such as FAA regulations of the United States. A pairing is infeasible if it violates agreements between employees, unions and employers. Sometimes, for the purpose of management operations, additional rules can be used to limit the number of feasible pairings. The airlines rules and regulations make the crew pairing problem quite complicated. It is quite hard to find a good method to deal with the diversity of rules and regulations. It is an area involving many researchers nowadays.

- Cost structure: it depends on the individual airlines. Some airlines pay a fixed salary for a working hour. Others calculate the salary based on credit hours which are the maximum of the flying time and some guaranteed minimum hours.

The cost of a pairing also includes a cost paid to the crew for staying away from their base (e.g., accommodations, transportation), *deadheading* (when crew are transported on a flight as passengers), and other penalty costs.

2.3 Literature Survey

Given the timetable of an airline company and the aircrafts associated with each flight, the *crew pairing problem* is to assign crews to service the flights such that the total service cost is minimal. Although we can see the crew pairing problem looks like the assignment problem, it has two inherent properties which make the problem more difficult. The first relates to a set of complicated rules specified by the airlines. Many of them are impossible or quite hard to model as mathematical formulations. The second concerns the complexity of the cost function which is often non-linear. Therefore, most successful published papers follow the idea of using an enumeration to consider all possible pairings. By this way, we can avoid the two mentioned difficulties in some extents.

The crew pairing problem is usually modelled as a *set partitioning problem*,

$$\text{(SPP)} \qquad \begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax = e \\ & x \in \{0,1\}^n. \end{aligned} \tag{2.1}$$

We denote the number of all feasible pairings by n. Every column of A represents a feasible pairing and every row corresponds to a flight that has to be serviced. Then, e denotes the unit vector which means each flight must be covered exactly by one pairing. Sometimes, the vector e is replaced by a vector of integer values which define how many crew members are required for a particular flight (see Andersson et al. (1997) for more details). We have $A_{ij} = 1$, if pairing j covers flight i, and $A_{ij} = 0$, otherwise. The cost of pairing j is given by c_j. The variable x_j states whether its respective pairing is in the optimal solution or not. The cost c_j of a pairing includes the crew costs, the accommodation costs and the penalty costs. In the following we do not distinguish between the terms "pairing", "column", and "variable".

With this way of transforming the crew pairing problem to the set partitioning problem, we will receive a huge number of pairings. Later in the thesis, readers will see that a timetable of 297 flights can generate millions of pairings. Besides the difficulties presented by integer constraints, the size of the constraint matrix is also an interesting target for research.

The example in Section 2.1 is now modelled as the following integer problem

$$
\begin{aligned}
\min \quad & c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 + c_5x_5 + c_6x_6 + c_7x_7 \\
& + c_8x_8 + c_9x_9 + c_{10}x_{10} + c_{11}x_{11} + c_{12}x_{12} + c_{13}x_{13} + c_{14}x_{14} \\
\text{s.t.} \quad & x_3 + x_6 + x_8 + x_{11} + x_{12} + x_{13} = 1 \\
& x_1 + x_2 + x_7 + x_9 = 1 \\
& x_3 + x_4 + x_{13} + x_{14} = 1 \\
& x_2 + x_5 + x_6 + x_7 + x_8 = 1 \\
& x_1 + x_9 + x_{10} + x_{11} + x_{12} = 1 \\
& x_5 + x_6 + x_{10} + x_{11} = 1 \\
& x_3 + x_4 + x_{13} + x_{14} = 1 \\
& x_4 + x_5 + x_{10} + x_{14} = 1 \\
& x_3 + x_6 + x_{11} + x_{13} = 1 \\
& x_1 + x_2 + x_7 + x_9 = 1 \\
& x_j \in \{0, 1\}, j = 1, \ldots, 14.
\end{aligned} \tag{2.2}
$$

A deadheading often happens when planners want to move crews around airports to satisfy a given timetable. Certainly, in that case, the deadheading will contribute certain cost to the pairing which contains the flight transporting the deadheading crew. This issue is modelled by changing the set partitioning form of (2.1) to the set covering form which means a flight leg can be covered by more than one pairing.

Among additional constraints, certain constraints can be presented directly within the integer model. One group of them is called *base constraints*, which restrict the aggregate time that all crews at a home base spend away from that home base. With these constraints, we can control the availability of crews at a base. They are modelled as the following inequalities:

$$t^0 \leq Bx \leq t^1,$$

where the rows of matrix B corresponds to the set of home bases. B_{ij} is the time that a crew spends away from base i for pairing j, and $B_{ij} = 0$ otherwise.

In this thesis, deadheading and base constraints will not be paid attention to. More reviews on them can be found in Rushmeier et al. (1995), Andersson et al. (1997). Next, we will review methods in the context of the crew pairing problem.

2.3.1 Pairing generation

It is reasonable not to consider all pairings of the whole schedule period for a large set of flight legs. Therefore, two general variants of pairing generation are easily seen: to generate all pairings for a subset of flights or to generate a limited number of pairings for the whole flight set.

Complete generation of a flight subset

This approach was originally developed by Rubin (1973) and has three main steps. First, a limited number of initial "good" feasible pairings are generated. Depending

on these pairings, we can obtain a better final solution even in a shorter computation time. This is typical for other local optimization methods. Generating the initial solution depends strongly on each specific crew pairing problem. One approach is to use the pairing construction methods of Baker et al. (1979) in this step. The set of flights building these initial pairings will be the base set for the second step. It is also called the subproblem. In the second step, *all* feasible pairings will be generated for the current subset of flights. Certainly, all labor rules and contractual regulations are taken into account in this step and the first step as well. The step results in a set partitioning problem. Third, the integer problem is solved by a set partitioning optimizer which can give a better objective cost. In this case, the new pairings will be used as the incumbent solution. It is quite clear that the method only finds a local minimum. In order to avoid it, before starting a new iteration, the method replaces some of the flight legs in the current set with ones which have not been covered yet. More heuristics are discussed in Anbil et al. (1991a). The process will go back to the first step if a certain terminating condition has not been reached.

In a paper, Andersson et al. (1997) say that the subproblem creation is not restricted to limiting the number of pairings. Planners can prefer different possibilities to build the subproblem which contains only a subset of flight legs. For example, limiting a time interval for pairings is a variant.

The complete generation for a flight subset has been chosen by most of major U.S. airlines. More details on this approach and its applications to real airlines can be found in Gerbracht (1978), Gershkoff (1989), Anbil et al. (1991a), Graves et al. (1993), Rushmeier et al. (1995), Wedelin (1995).

Partial generation of a full flight set

In a different way, the second idea is to generate a limited number of pairings considering the whole set of flight legs. Working directly on a very large number of pairings is not a good choice. Instead, the approach employs techniques to pick up only favorite pairings. The approach has three main steps. The first again is to generate an initial solution which consists of "good" pairings. In the second step, the subproblem which has been created by the current set of pairings is solved. Basing on information of the optimization, the third step will generate favorite pairings in order to be put into the current subproblem. A widely-used method in this step borrows an idea from linear programming, called *column generation*. It chooses columns (pairings) which have a negative reduced cost in the expectation of improving the cost function. By doing that, we do not have to make an exhaustive search through all feasible pairings. The problem of the third step now can be modelled as a constrained network problem (Desrosiers et al., 1995).

The second approach with column generation method has been a well-known method for many routing and scheduling problems in recent years. Desrosiers et al. wrote a good survey on this, not only in theoretical aspects but also in applications. Barnhart et al. (1998) also provide us with an interesting survey on branch-and-price, which is a

combination of column generation and the branch-and-bound framework. Applications of this approach in the airline crew pairing problem can further be found in Lavoie et al. (1988), Desaulniers et al. (1997a), Vance et al. (1997), Gustafsson (1999).

2.3.2 Methods

Two schemes presented in the previous section is the ways to reduce the problem size if working on original problems is impossible. They both reduce these crew pairing problems to smaller ones. Whenever the reduced subproblems can be solved in terms of memory and computing power, they can be handled by methods to be surveyed next.

Branch-and-cut-and-price based methods

Branch-and-cut has been considered as a good framework for general mixed integer solvers. Nowadays, there are many libraries following this direction and they are also used to solve integer problems coming from the crew pairing problem. However, a general solver cannot be much efficient to handle this specific problem. Therefore, a special solution should be required. Hoffman and Padberg (1993) present one of the most successful works on the crew pairing problem in this context. The branch-and-cut framework has been chosen to solve set partitioning problems transformed from crew pairing problems. Their method and set of test problems have become well-known in this research area. Certainly, since they only focus on set partitioning problems, issues relating to real airlines operations have been dropped out of consideration. Borndörfer (1998) revisits three closely related problems: set packing, set partitioning, and set covering problems. Then, he also contributes a fast algorithmic implementation to solve the problems of Hoffman and Padberg. Computational results are impressive. Many problems can be solved using only the cutting plane phase without any further branching. We will see more details about this approach in Chapter 3.

Branch-and-price solution methods follow the idea mentioned in the previous section. The main difference to branch-and-cut is to generate variables instead. By this way, they possibly work with large problems. The main difficulty of this method belongs to a subproblem which readers will see more details in Chapter 4.

LP-based heuristics

There are many heuristics for set covering problems which can come from crew pairing problems. Only some of them are listed here due to their successes. Fisher and Kedia (1990) apply a heuristic to improve the lower bound. The well-known 3-opt operation in combinatorial optimization is used by these authors to modify the dual solution when keeping the dual feasibility. Techniques in set covering problems by Chvátal (1979) are chosen for the primal heuristics. The algorithm used by Fisher and Kedia also includes a branch-and-bound algorithm in case that the primal and dual bound are not equal.

Another good heuristics applied to airline scheduling is the cost perturbation of Wedelin (1995). Let's consider the Lagrangian relaxation of (SPP)

$$\min_{0 \leq x \leq 1} c^\top x + \pi^\top (b - Ax). \tag{2.3}$$

It is clear that if the solution to the relaxed problem is feasible to (SPP), then it is also optimal for (SPP). Wedelin contemplates the impact of the reduced cost vector $\overline{c} = c - \pi^\top A$ on the solution x as following:

$$x_j = \begin{cases} 0 & \text{if } \overline{c}_j > 0 \\ \text{any value in } [0,1] & \text{if } \overline{c}_j = 0 \\ 1 & \text{if } \overline{c}_j < 0 \end{cases} \tag{2.4}$$

In the process of solving a Lagrangian dual of the relaxation, the author makes an attempt to perturb the vector c in order to satisfy the fact that reduced costs are greater than 0 or less than 0 for all j. The solution method has been applied in a software package for airline management operations (Andersson et al., 1997).

Vance et al. (1997) report an LP-based heuristic which uses many techniques in branch-and-price. The authors have decided to solve the column generation problem approximately, making the whole algorithm find only near optimal solutions. Furthermore, the tree is just explored partly. However, it is quite easy to turn it to an exact method. We will return to this method in Chapter 4.

Genetic algorithms

Genetic algorithms are also involved as tools to solve the crew pairing problem (Chu and Beasley, 1995, Levine, 1996). Although being not more successful than the traditional approach using the branch-and-bound framework, they still have some advantages, such as keeping a population of possible solutions. From a practical point of view, this is preferable. However, as shown in the computational results of the papers above, the performance of this approach is quite poor. When solving a standard set of set partitioning problems generated from airline crew pairing problems, genetic algorithms only present optimal solutions for a subset. Moreover, because of working on set partitioning problems, the genetic algorithms still do not consider the difficulties from complex airline rules and regulations.

Tabu search

One of few papers on tabu search in the area of crew scheduling is by Cavique et al. (1999). In their work, the solution for an urban transportation problem has been solved by techniques in Tabu search. Moreover, since the rules on crew schedules are extremely strict, the authors employ a technique to separate the neighborhood structure into two substructures to deal with the easy in-feasibility. The ejection chains technique (Rego, 1998) has been used in order to reduce the neighborhood. The approach is proved to be effective for solving a set of crew scheduling problems in urban transportation.

Constraint logic programming

Constraint logic programming (CLP) has been revisited in recent years due to its ability of expressing complex working rules which are quite hard to be presented by operations research (OR) techniques. In addition, nowadays there are more and more rules which cannot be modelled in a linear form because they do not satisfy certain properties (see Desrosiers et al., 1995). Another reason for constraint programming's comeback is its improved performance. With constraint propagation and domain reduction mechanisms, a constraint programming environment can spread a variable bounding throughout the whole space, reducing the search space of feasible solutions. Clearly, it is more comfortable to use the declarative programming model of constraint programming to represent airline rules. There are some efforts following a natural approach to use constraint programming completely to solve the problem. However, this approach seems not to be so successful. The performance of the inference engine in constraint programming tools is still poor (Guerinik and Caneghem, 1995). Thus, techniques from operations research are still highly efficient to support constraint programming, introducing a hybrid approach. Roughly speaking, there are two ways of integrating operations research and constraint programming: OR in CLP framework and vice versa. The latter is often the preferred one. A good paper on a general framework to apply CLP in a column generation algorithm is presented by Fahle et al. (1999) for crew assignment problems. Yunes et al. (2000) show another real-world application of this approach for a crew scheduling problem of a metropolitan area.

Other heuristics

One of the early methods for the crew pairing problem was presented by Baker et al. (1979). Carefully considering characteristics of an airline, the paper shows a pairing construction procedure which is a continuous process to assign flight legs to a partially completed pairing. Certainly, there are several criteria for assignment. The process is ended when the pairing reaches the starting base. Then, one of the remaining flight legs is picked as a seed of the new pairing in the next iteration. The method is simple to implement and can be used to construct an initial solution for other methods.

Readers will find more useful information on this approach towards the crew pairing problem in papers of Barnhart et al. (1997), Lagerholm et al. (2000), Klabjan et al. (2001b,a, 2002).

Stochastic programming

The aviation industry is a very dynamic environment. The real airline operations can be different with the planned schedule due to disruptions. For example, a delay in the arrival of one flight possibly breaks the original schedule. To overcome this kind of problems, planners often perform a recovery procedure to bring the operations back to the original schedule. This is not easy but rather ineffective. Therefore, a robust solution which can compensate for the early disruption is quite important in airline

operations. This means the solution still minimizes the cost function, and be less sensitive to the schedule disruptions. Stochastic programming is often the choice of researchers to deal with the uncertainty where the probability distribution of unknown data is known or can be estimated. Methods in this area often follow the two-stage stochastic integer programs with the recourse. One of the model can be as following,

$$\begin{array}{ll} \min & c^\top x + \sum_{k=1}^{K} p^k q^\top y^k \\ \text{s.t.} & Ax \leq b \\ & Tx + Wy^k = b^k, k = 1, \ldots, K \\ & x \in X, y^k \in Y, k = 1, \ldots, K \end{array} \tag{2.5}$$

where K is the number of scenarios, and p^k is the probability of the k-th scenario. The modified model (2.5) has an additional expected cost for disruptions. Benders' decomposition is a well-known approach for solving this class of problems. Note that the objective function now changes to minimizing the *expected* cost. Some works in this direction are Yen (2000), Schaefer (2000).

Parallel optimizer

Generally, there are two ways of solving the crew pairing problem in parallel. The first is to study the integer model of the problem and to solve it as a general integer model. This approach has been performed as a generalized parallel mixed integer optimizer or a parallel set partitioning optimizer. Using strong cutting planes for the set partitioning problem is often better than using general mixed integer cuts. One work in this direction is the parallel set partitioning solver of Linderoth et al. (2001). Besides employing recent techniques for the set partitioning solver, the authors apply both the functional decomposition and domain decomposition for parallel environment. The parallelization is quite complicated and quite specific to the set partitioning problem. Computational results of the work prove that the parallel computing is effective with large scale combinatorial optimization problem, even though the speedup is not quite good.

The second approach considers the crew pairing problem itself directly. Alefragis et al. (2000) work on the heuristic presented by Wedelin (1995), and build a parallel optimizer for crew scheduling problems. As seen above, the sequential heuristic depends strongly on variable updates to solve a Lagrangian dual problem. Therefore, in order to separate computing tasks among processors, the presented method uses a fined grained parallelism, which distributes variables across the processors. The paper shows good speedups with a small number of processors in solving real world airline crew pairing problems. Some more details on the integration of this work to a commercial software (named PAROS) can be found in Alefragis et al. (1998). In PAROS, the parallelization is not implemented in the process of solving the linear relaxation problem of the master problem. Instead, it distributes the enumeration of pairings over the processors and their outputs will be sent into the parallel set partitioning optimizer mentioned before.

To support other researchers, Klabjan and Schwan (1999) have made a research on

how to generate pairings quickly on a cluster of machines. As shown in their work, and in this thesis as well, there are often a very large number of feasible pairings for each flight set. This requires a method to work efficiently. Workload balancing schemes are introduced to guarantee a fair distribution of tasks among processors in the process of exploring the whole set of pairings.

Over several decades, it is obvious that there are many research efforts on the problem. The survey above on this area may not include all of them. However, there are still many gaps for the research. In the next chapters, the research on the airline crew pairing problem will be presented. The motivation will be presented in each of them. Main contributions of this thesis will be presented in Chapters 4, 5, and 6. In the next section of this chapter, the thesis will present an airline crew pairing problem which serves as a main application throughout the thesis.

2.4 A Case Study: Vietnam Airlines

The crew pairing problem in Vietnam Airlines is the first problem which attracted me in the beginning of the Ph.D. research. Until 2002, the problem has been still solved manually by a group of personnel in a crew scheduling department. Since the number of flights is small and the operational flight network is simple, this task can be done in such a way. Furthermore, the planners have only focused on the availability of crew for flight legs within an "acceptable" cost . In other words, a solution for the crew pairing problem in Vietnam Airlines has been only ensured to be feasible. The cost function has not been paid much attention although it still has been considered intuitively in the manual scheduling process. Below, only information allowed to obtain on the operations of this company will be presented. Figure 2.2 sketches functional blocks of the scheduling process in Vietnam Airlines, including both crew pairing and crew rostering.

Characteristics

Under the point of view presented in Section 2.2, the main aspects of the crew pairing process in Vietnam Airlines are introduced as following:

- Crew category and fleet: the pairing problem is decomposed by crew category. As mentioned before in other airlines, the crew pairing in Vietnam Airlines is also decomposed by crew category due to the difference of properties. Pilots are qualified to fly one aircraft type. Cabin crew can fly several aircraft types. And each person in crew planning group of Vietnam Airlines performs scheduling for each aircraft type in a manual process. Table 2.1 shows different aircraft types along with their number of flight legs in a week. In summary, the crew pairing in Vietnam Airlines is decomposed by crew category and aircraft type.

- Regularity of timetable: Vietnam Airlines operates a constant schedule for every week within a session (a half of year). Column "#daily legs" of Table 2.1 says

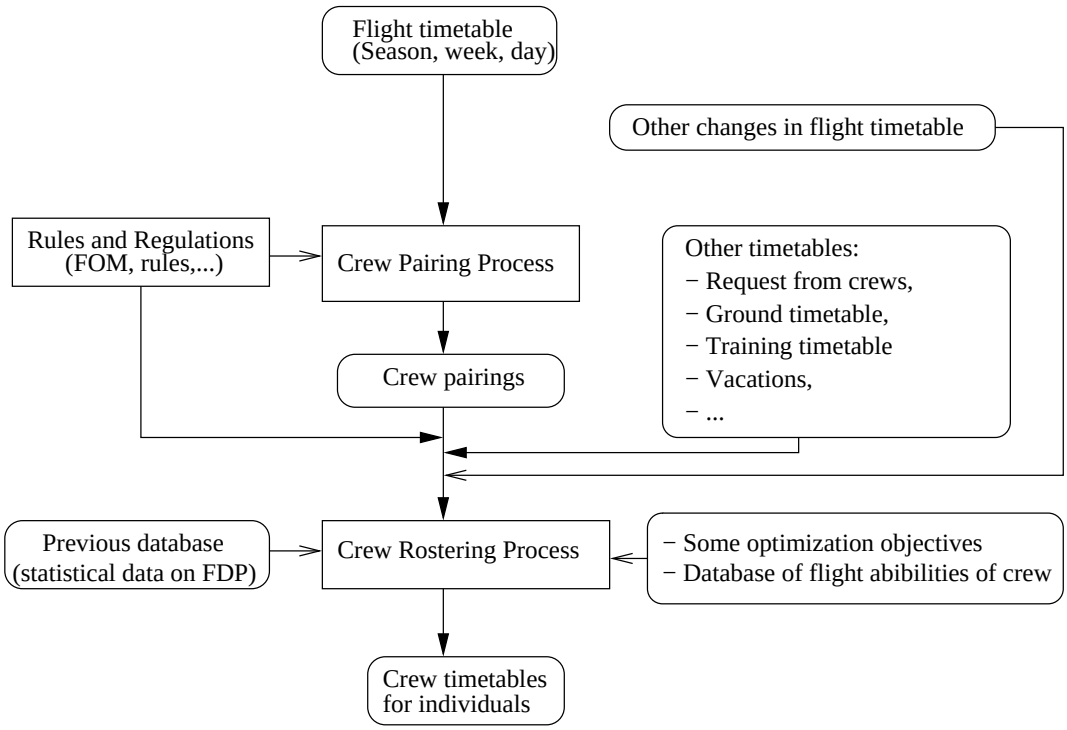


Figure 2.2: Scheduling steps in Vietnam Airlines

Name	Aircraft type	#legs per week	#daily legs
vn320	320 (Airbus 320)	301	140
vn321	321 (Airbus 321)	168	126
	330 (Airbus 330)	12	0
	343 (Airbus 340-300)	6	0
	734 (Boeing 737-400)	8	0
	737 (Boeing 737-100)	34	0
	763 (Boeing 767-300)	86	14
	767 (Boeing 767)	4	0
	76C (Boeing 767-300 freighter)	2	0
	777 (Boeing 777)	58	7
vnAT7	AT7 (ATR 72)	296	210
	F70 (Fokker 70)	56	56
total		1031	553

Table 2.1: Aircraft types and their numbers of flight legs (Sep 6, 2004 – Sep 13, 2004)

that about one half of all flight legs are operated every week day. They often fall into categories: short and medium-range connections. Generally, it can be said that the company applies a weekly schedule.

- Network structure: the operational flight network of Vietnam Airlines is quite simple (for each kind of category). In Figure 2.3, you can see the networks of Vietnam Airlines, in which flights often start from SGN or HAN to a subsidiary airport and return to them immediately after that.

  Note that the scaling ratio of sub-figures are not the same. Moreover, the positions and arc lengths do not correspond to real locations either.

- Rules and regulations: the scheduling process must follow strictly the *flight operations manual* (FOM) (Vietnam Airlines, 1998). Several rules which have impact on the crew pairing process have been collected and they are presented as following:

  - Flight duty period shall not exceed in times in the planning table 2.2, according to the current number of flight legs that have been taken.

Reporting time (LT)	1,2	3	4	5	6 or over
07:00-17:59	1330	1230	1200	1100	1030
18:00-21:59	1330	1200	1130	1030	1000
22:00-04:59	1300	1130	1100	0930	0900
05:00-06:59	1330	1200	1130	1030	1000

Table 2.2: Flight duty limitation

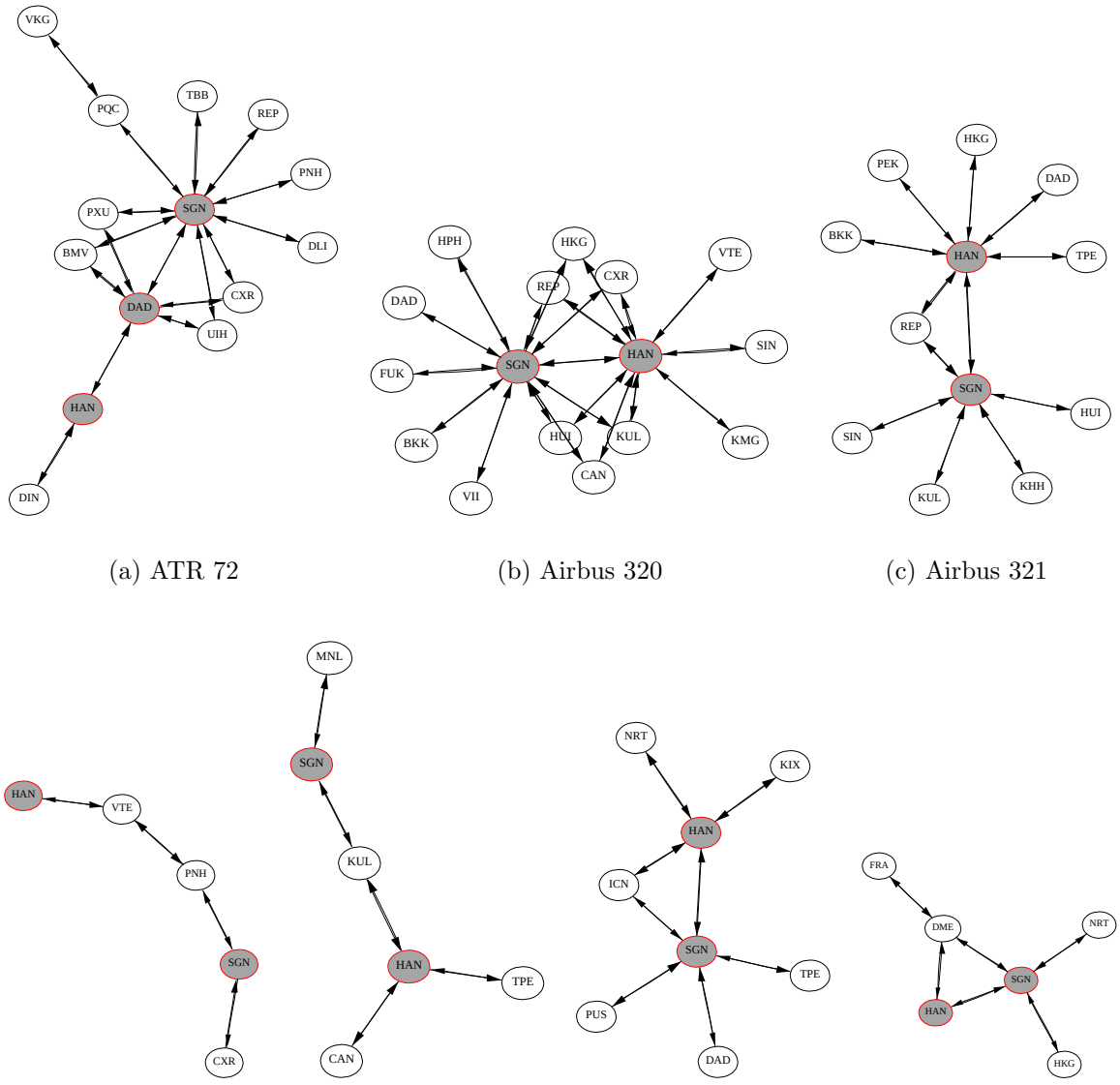
    The numbers in the second column and columns after it mean the time duration (e.g., 1330 means 13 hours and 30 minutes).

  - The minimum rest time period which must be provided before undertaking a flight duty period shall be at least as long as the preceding duty period, and not less than 11 hours. In case of time zone difference, rest periods would be longer (more details in FOM).

  - Some other control variables:

    * Pre-flight and post-flight-time,
    * Minimum ground time (minimum time between two legs that can be connected together in a pairing),
    * Maximum number of days for a pairing,
    * Maximum number of flight legs in a pairing,

- Cost structure: Due to its strictly confidential nature, this is the most unknown part in Vietnam Airlines. As mentioned in the beginning, the planners often try to find a "good" feasible solution. The scheduling process does not pay much attention to finding an optimal or near-optimal solution. Vietnam Airlines only distinguishes 3 kinds of objectives: a cost for flying time, the crew utilization and



(a) ATR 72 (b) Airbus 320 (c) Airbus 321

(d) Fokker 70 (e) Boeing 734, 737, and 738 (f) Boeing 763, 767, and 76C (g) Boeing 777

Figure 2.3: Operational flight network for each aircraft category. (Note that home bases are denoted by shaded ellipses.)

a cost for ground time. The company pays each crew a fixed amount of money for a unit of flying time. Therefore, with a given timetable, the company must spend a fixed cost to cover all flights. The crew utilization and the resting time are the two most important factors in the planning. But, the planners do not realize how important they are. Therefore, an assumption must be made that the most important cost which Vietnam Airlines wants to minimize is the cost for ground time which is mainly the cost for the accommodation when crews take a rest far away from their bases. In this thesis, the deadheading and some other penalty costs will not be taken into account.

Applied methods

Algorithm 2.4.1 Tung's heuristic for the crew pairing problem

```
p ← first flight starting from a home base.
while (p is not valid) do
  d ← a schedule direction based on p.
  F ← list of flight legs which can be appended to p and with the direction d.
  if ( F == ∅ ) then
    break
  end if
  f ← the best first of F.
  p ← p + f.
end while
```

To my knowledge, there is only one work on this problem at Vietnam Airlines. More details can be seen in the work of Tung (1998). Outline of the suggested heuristic is shown in Algorithm 2.4.1.

The algorithm is based mainly on the idea of Baker et al. (1979). Furthermore, it is quite simple and cannot ensure the quality of the final solution. The only difference is to use the variable d to specify the preferred direction to be scheduled. The direction here means the crew should "leave" or "return" the home base where they start. In that work, this has been done intuitively because the author has experiences in Vietnam Airlines. There is no algorithmic method suggested in this part.

In Table 2.1, the three largest sets of flights legs belong to the aircrafts Airbus 320, Airbus 321, and ATR 72. We will use these problem sets as case studies in the crew pairing solvers presented in the next chapters. Column "Name" of Table 2.1 denotes the problem names.

Chapter 3

A Branch and Cut Approach

3.1 Set Partitioning Model

It can be said that set partitioning is one of the most widely used models in practical applications.

Definition 3.1.1. A *set partition* problem is defined as following:

$$\text{(SPP)} \qquad \begin{array}{rll} \overline{z}_{\text{SPP}} = & \min & c^\top x \\ & \text{s.t.} & Ax = e \\ & & x \in \{0,1\}^n, \end{array} \tag{3.1}$$

where A is an $m \times n$ matrix of zeros and ones, c is an arbitrary n-vector. Note that the right hand side e is an m-vector of 1's.

Let $M = \{1, \ldots, m\}$ be the row index set and $N = \{1, \ldots, n\}$ be the column index set of (3.1). For each column $A_{\cdot j}$, let $M^j = \{i \in M : A_{ij} = 1\}$ and associate the set with a cost c_j. We also denote by $N^i = \{j \in N : A_{ij} = 1\}$. With these notations, we can interpret a (SPP) as finding a minimum cost family of subsets $M^j, j \in N$ which is a partition of M. It is not true that every subset of M can belong to the optimal partition. This relates to the fact that "application" constraints seems to be embedded into the definition of the set $\{M^j : j \in N\}$.

A large number of scheduling problems can be formulated as follows: given

(i) a finite set M,

(ii) a constraint set C defining a family $\mathcal{P}$ of "*feasible subsets*" of M, and

(iii) a cost associated with each member of $\mathcal{P}$,

find a minimum cost collection of members of $\mathcal{P}$, which is a partition of M.

In order to solve such scheduling problems, we can use a two-step framework described as following:

Step 1: Generate *explicitly* all feasible subsets of M according to the constraint set C (i.e., generate $\mathcal{P}$). An approximation approach could be performed by creating a subset $\mathcal{P}' \in \mathcal{P}$ so that the probability of an optimal solution being contained in $\mathcal{P}'$ is sufficiently high.

Step 2: Create a (SPP) in which $\mathcal{P}$ ($\mathcal{P}'$, for approximations) defines the set of columns. Solving the (SPP) is equivalent to solving the original scheduling problem.

Applying the idea above, the set partitioning problem is used to model many practical applications: bus crew scheduling, airline crew scheduling, vehicle routing, circuit design, facility location problems, etc.

3.1.1 Set packing and set covering models

Replacing the equality constraints with inequality ones, we obtain two closely-related problems. They are the *set packing* problem:

$$\text{(SP)} \qquad \begin{aligned} \overline{z}_{\text{SP}} = \quad & \min \quad c^\top x \\ & \text{s.t.} \quad Ax \leq e \\ & \qquad x \in \{0,1\}^n, \end{aligned} \tag{3.2}$$

and the *set covering* problem:

$$\text{(SC)} \qquad \begin{aligned} \overline{z}_{\text{SC}} = \quad & \min \quad c^\top x \\ & \text{s.t.} \quad Ax \geq e \\ & \qquad x \in \{0,1\}^n. \end{aligned} \tag{3.3}$$

We also define six associated polyhedra:

$$\begin{array}{ll} P_{\text{SPP}}(A) = \text{conv}(x \in \{0,1\}^n : Ax = e) & P_{\text{LSPP}}(A) = \{x \in \mathbb{R}^n_+ : Ax = e\} \\ P_{\text{SP}}(A) = \text{conv}(x \in \{0,1\}^n : Ax \leq e) & P_{\text{LSP}}(A) = \{x \in \mathbb{R}^n_+ : Ax \leq e\} \\ P_{\text{CP}}(A) = \text{conv}(x \in \{0,1\}^n : Ax \geq e) & P_{\text{LCP}}(A) = \{x \in \mathbb{R}^n_+ : Ax \geq e\}. \end{array}$$

At a first glance, (SP) is a tightly constrained problem, and (SC) is a loosely-constrained problem, in comparison with the set partitioning problem. In a (SC), the set M is likely to be divided into *overlapping* subsets. Meanwhile, a (SP) restricts that a member of M only belongs to at least one subset.

Without loss of generality, it is assumed that there are no empty columns and rows in A which can be redundant, making problems unbounded, or infeasible. Obviously, $P_{\text{SPP}}(A) = P_{\text{SP}}(A) \cap P_{\text{SC}}(A)$. With this observation, working on the set partitioning model could be done by investigating equivalent models (SP) and (SC). Lemke et al. (1971) shows how to transform a (SPP) to (SC). However, (SC) cannot be brought to a (SPP). On the other hand, (SP) and (SPP) are equivalent (i.e., two problems have the same optimal solutions) under certain conditions. In more details, if we denote by $c' = \theta eA - c$, (SPP) can be rewritten as a (SP) as following:

$$\begin{array}{ll} \max & \theta m + {c'}^{\top} x \\ \text{s.t.} & Ax \leq e \\ & x \in \{0,1\}^n. \end{array}$$

Whenever the new (SP) is feasible, and θ, a scalar, is sufficiently large, the new problem has the same optimal solution as the (SPP). This equivalence is important because instead of considering the original problem, we can investigate the related problems.

Theory on set packing and set covering shows a strong relation of them to associated problems in graph theory. Given an undirected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. Constructing a node-edge incidence matrix A^G (i.e., $A^G_{ij} = 1$ if node i is incident to edge j). An *edge matching* $M_E \subseteq E$ is a set of disjoint edges, that is, at most one edge of a matching is incident to any node $v \in V$. A matching is called *perfect* if every node of G is incident to exactly one edge in the matching. Oppositely, we have a definition of an *edge covering* which is a set $C_E \subseteq E$ of edges being incident to every node of G. This also means every node of G is incident to at least one edge of C_E. The problem of finding maximum cardinality edge matching (edge matching problem) is

$$\text{(EM)} \qquad \begin{array}{lll} \overline{z}_{\text{EM}} = & \max & e^{\top} x \\ & \text{s.t.} & A^G x \leq e \\ & & x \in \{0,1\}^n, \end{array} \tag{3.4}$$

and the problem of finding minimum cardinality edge covering (edge covering problem) is

$$\text{(EC)} \qquad \begin{array}{lll} \overline{z}_{\text{EC}} = & \min & e^{\top} x \\ & \text{s.t.} & A^G x \geq e \\ & & x \in \{0,1\}^n. \end{array} \tag{3.5}$$

A *node packing* is a set $M_N \subseteq N$ of nodes such that every edge of G is incident to at most one node of M_N. A *node covering* is a set $M_C \subseteq N$ of nodes such that every edge of G is incident to at least one node of M_C. There are also two associated problems which are to find the maximum cardinality node packing (node packing problem) defined as follows:

$$\text{(NP)} \qquad \begin{array}{lll} \overline{z}_{\text{NP}} = & \max & e^{\top} x \\ & \text{s.t.} & {A^G}^{\top} x \leq e \\ & & x \in \{0,1\}^n, \end{array} \tag{3.6}$$

and to find the minimum cardinality node covering (node covering problem) defined as follows:

$$\text{(NC)} \qquad \begin{array}{lll} \overline{z}_{\text{NC}} = & \min & e^{\top} x \\ & \text{s.t.} & {A^G}^{\top} x \geq e \\ & & x \in \{0,1\}^n. \end{array} \tag{3.7}$$

There is an interesting relation among the optimal solution of these problems that was presented by Gallai (1959).

Theorem 3.1.2. *For any nontrivial connected graph G with n nodes,*

$$\overline{z}_{\mathrm{EM}} + \overline{z}_{\mathrm{EC}} = n = \overline{z}_{\mathrm{NP}} + \overline{z}_{\mathrm{NC}}$$

Although (EM) and (EC) can be solved by a polynomially bounded algorithm, (NP) and (NC) are classified as $\mathcal{NP}$-hard problems (assume $\mathcal{NP} \neq \mathcal{P}$). If we create an *intersection graph* $G^A = (N, E)$ of A which has one node for every column of A, and one edge for every nonorthogonal column of A (i.e., $(i, j) \in E$ if and only if $A_{\cdot i} A_{\cdot j} \geq 1$. Denote by A^G node-edge incident matrix of G^A. A weighted node packing problem in which the objective vector e is replaced by the objective vector c is

$$\begin{array}{lll} \overline{z}_{\mathrm{WNP}} = & \max & c^\top x \\ \text{(WNP)} & \text{s.t.} & {A^G}^\top x \leq e \\ & & x \in \{0,1\}^n. \end{array} \tag{3.8}$$

If (WNP) is feasible, the corresponding (SP) is also feasible and $\overline{z}_{\mathrm{WNP}} = \overline{z}_{\mathrm{SP}}$. The converse is true as well. This remark gives a way to solve a set packing problem by solving a node packing problem which has a more structured constraint matrix.

The set packing problem also relates closely to the *stable set problem* on the associated intersection graph with node weights c_j. From this point of view, a (SP) is merely to find the maximum weight of stable sets which are defined as the sets of pairwise nonadjacent nodes.

In the thesis, a further review on related problems will not be included. Readers could refer to Edmonds (1965) for the original work on edge matching problems. A more general review is of Nemhauser and Wolsey (1988, Chap. III.2). Since being classified as an $\mathcal{NP}$-hard problem, (NP) has attracted many researchers as for finding facets and algorithms to separate them. Some of them are Padberg (1973), Edmonds (1965), Trotter (1973), Grötschel et al. (1988). There are some good surveys, such as Balas and Padberg (1976), Borndörfer (1998).

3.1.2 Crew pairing to set partitioning problem

As presented in Chapter 2, crew pairing is a scheduling problem, and certainly can be transformed into a (SPP) using the two-step idea mentioned in Section 3.1. Assume a flight set $\mathcal{F} = \{f_i : i = 1, \ldots, m\}$ for a schedule period. This set corresponds to M in the method. Following a set of rules and regulations, a set of all feasible pairings $\mathcal{P} = \{p_i : i = 1, \ldots, n\}$ is generated for that period. Note that, in the thesis, a schedule is assumed to be repeated for next periods (e.g., a weekly schedule will be repeated exactly the same during the following weeks). Changes in schedule in real airline operations will not be considered in the thesis as mentioned in Section 2.2.

An example should explain the idea more clearly. Remember the example in Chapter 2. Under a given set of airline rules, the first step gives us the set partitioning problem (2.2).

We can formulate the crew pairing problem with variables x_{ij} where $x_{ij} = 1$ if crew i is assigned to flight leg j. However, this way of formulating produces a highly non-linear objective function which is difficult to express. Therefore, this is also another advantage of applying the set partitioning model when dealing with non-linear cost functions. Cost for a pairing depends on airlines and is easily calculated while generating pairings. The cost c_j of a pairing possibly includes the crew costs, the accommodation costs and the penalty costs. After all, a (SPP) has been created in which each column is a pairing and the objective is to partition all flight legs into a set of pairings with a minimum cost.

Although the idea is quite simple, there are issues which should be described in more details.

Pairings over schedule boundary

Although we only consider the time period of a given schedule, there still exist some pairings starting in this period and stopping in the successive one. In the example, the schedule is daily, several pairings start in the current day but stop in the next day, denoted by a line from the rightmost flight to the leftmost flight. This means, in order to generate all feasible pairings, we need to duplicate the *ground* flight set to the next periods. The extent of such a duplication depends on the maximum length of a valid pairing. Note that flight legs which are repeated for consecutive periods are considered as the same in the process of transforming into a (SPP) (i.e., they correspond to the same row of the problem). In the example above, such pairings are p_6, p_7, p_8, p_9 and p_{10}.

Large number of feasible pairings

It is quite clear that there are a lot of pairings generated in spite of the limitation on the number of flight legs in a pairing and restrictions of airline rules as well. Exhaustive enumeration is combinatorial explosive which should be considered carefully. From a practical point of view, the more pairings, the more difficulties there are. Then, the more memory is required to store them. Furthermore, a data structure to keep all feasible pairings in memory can impact the efficiency of algorithms (see Hu and Johnson, 1999). The thesis will return to this topic later in computational issues. A (SPP) with a very large number of columns could be a difficult zero-one optimization problem. Mentioned in many real applications, the number of all pairings can reach billions for a set of about 1000 flight legs. Therefore, crew pairing generation is also a time consuming task which even needs support from parallel computing (Klabjan and Schwan, 1999).

There are generally two approaches to avoid dealing directly with a large set of pairings. The first one was described by Rubin (1973), Gerbracht (1978) and implemented in the software package ALPPS. The method starts by choosing randomly a subset $\mathcal{F}'$ of $\mathcal{F}$ (to satisfy some condition, e.g. $|\mathcal{F}'|$ is less than a given number) and generate the set $\mathcal{P}'$ of *all* possible pairings for it. A new (SPP) in which the set of

rows corresponds to $\mathcal{F}'$ and the set of columns corresponds to $\mathcal{P}'$ will be created and solved by a set partitioning solver. Columns in the incumbent solution (if it exists) are kept in $\mathcal{F}'$. The next iteration again randomly selects flight legs for $\mathcal{F}'$. The process above is iteratively executed until a limitation of time is reached or there is no further improvement in the objective value. A detailed procedure of this approach can be found in Anbil et al. (1991a).

An alternative approach is to initially generate a set $\mathcal{P}'$ of pairings which covers *all* flight legs of $\mathcal{F}$. There is a (SPP) corresponding to these pairings and flight legs. Only pairings belonging to the final solution (if it exists) are kept in $\mathcal{P}'$ which will again include additional "good" pairings generated by a heuristic process. A "good" pairing is possibly determined by the value of its corresponding binary variable in linear relaxation (e.g, $x_p > 0$). Two steps of generating pairings and optimizing set partitioning problems are repeated until some given condition is satisfied. This approach is implemented in TRIP, a software package to solve crew scheduling problems for American Airlines. A report of Anbil et al. (1991b) describes a successful attempt to work with a real practical problem. Another paper applying this approach was presented by Chu et al. (1997).

Both approaches require a set partitioning optimizer. Generally speaking, an outer loop of including more flight legs (first approach) or generating more pairings (second approach) is just a way of avoiding being kept in local optima. Therefore, solving set partitioning problems efficiently is the most important topic in this area. In the next sections of the chapter, aspects of polyhedral theory of (SPP) will be considered under a practical point of view. Readers being interested in two approaches above can refer to Andersson et al. (1997), Hoffman and Padberg (1993), Gustafsson (1999) for more descriptions.

Density of constraint matrix

The example before shows that the constraint matrix is very sparse. In the crew pairing step, a crew is often prohibited to fly more than a given number of flight legs which is small with respect to the total number of flights. For example, in Vietnam Airlines, the former number is set to 6. This also means, under specific rules, the more flights to be scheduled, the more sparse the constraint matrix. The well-known test problems supplied by Hoffman and Padberg (1993) support this comment. More experiences on the density of the constraint matrix can be found in Borndörfer (1998). The density of the test problems will be given in Section 3.3. Considering the sparsity of the constraint matrices should be helpful in organizing a good storage structure.

3.2 Facial Structure of Set Partitioning Polytope

As mentioned in a remark of Section 3.1.1, the investigation on the polyhedron $P_{\mathrm{SPP}}(A)$ can be done by studying the two associated polyhedra $P_{\mathrm{SP}}(A)$ and $P_{\mathrm{SC}}(A)$. Following

sections will consider well-known facets which were discovered and are widely used in practical solvers.

3.2.1 Set packing facets

Facets of set packing polytopes are often obtained from some subgraphs of the intersection graph. These subgraphs are called *facet defining graphs*. We will introduce the facets with their associated graphs.

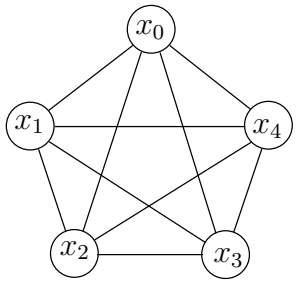
Clique inequality

Consider the intersection graph $G = (N, E)$ of a (SP) ($N = \{1, \ldots, n\}$). A clique in the graph G is a maximal complete subgraph of G.

Theorem 3.2.1. *An inequality*

$$\sum_{i \in C} x_i \leq 1, \tag{3.9}$$

where $C \subseteq N$, is a facet of P_{SP} if and only if C is the node set of a clique in G (Fulkerson, 1971, Padberg, 1973).



It is quite clear that (3.9) is valid for all $x \in P_{\text{SP}}$ because C is the node set of a clique. We create n linearly independent zero-one solutions to (SP): for each $i \in C$, set $x_i = 1$ and $x_k = 0$ for $k \in N \setminus \{i\}$ (obtaining $|C|$ solutions); for each $i \in N \setminus C$, choosing $j \in C$ so that $(i, j) \notin E$ (there exists at least such j because C is the node set of the *maximal* complete subgraph), set $x_i = x_j = 1$ and $x_k = 0$ for all $k \in N \setminus \{i, j\}$ (obtaining $|N \setminus C|$ solutions). These n solutions are valid to P_{SP} and satisfy (3.9) with equality. Therefore, (3.9) is a facet and the "if" direction has been proved.

The "only if" direction can be seen as follows. Suppose a graph of C and edges which join them in G is not a maximal complete subgraph. This means that there exists i such that $C \cup \{i\}$ with the respective edges of G make a larger complete subgraph. We can choose n linearly independent solutions to P_{SP} which satisfy (3.9) with equality. Thus, they also satisfy

$$\sum_{k \in C \cup \{i\}} x_k \leq 1$$

with equality. Furthermore, we can create at least one solution $x_i = 1$ and $x_k = 0$, otherwise. The new solution is clearly independent from the others. It is a contradiction because there are not more than n linearly independent points on a hyperplane of $\mathbb{R}^n$.

Odd cycle inequality

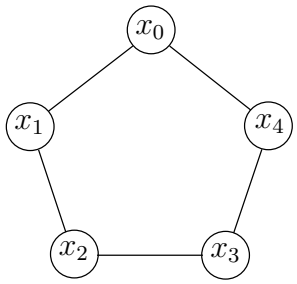
Given the intersection graph $G = (N, E)$ of a (SP), an edge connecting two non-consecutive nodes of a cycle of G is called *chord*. We have a theorem below about

another valid inequality.

Theorem 3.2.2. *Given an odd cycle C in G, the inequality*

$$\sum_{i \in C} x_i \leq \frac{|C| - 1}{2}, \tag{3.10}$$

where C is the node set of the cycle, is a valid inequality to (SP).



If x is feasible to (SP), it is clear that we can find $j \in C$ with $x_j \leq 1/2$. Without loss of generality, we can number the node set of the odd cycle C by $1, \ldots, 2k+1$, and assume $j = 2k + 1$. We have

$$\begin{aligned} \textstyle\sum_{i \in C} x_i &= \textstyle\sum_{i=1}^{k} (x_{2i-1} + x_{2i}) + x_{2k+1} \\ &\leq k + \tfrac{1}{2} \end{aligned}$$

because the sum of the objective values of each pair of consecutive nodes is less than 1. Since $x_i \in \{0, 1\}$, we can drop the fraction to obtain (3.10).

Generally, an odd cycle inequality is not a facet. However, Padberg (1973) showed in a paper that an odd cycle without chord can be lifted to a facet of the polytope P_{SP} by a procedure called *sequential lifting* (see Hoffman and Padberg (1993), Borndörfer (1998) for more details). However, this procedure is too computationally expensive to be implemented. This is the reason why the thesis does not consider it further.

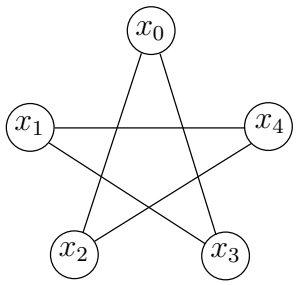
Other inequalities

Set packing polytopes are studied well among many combinatorial optimization problems. Therefore, many their facets have been found so far. Some of its valid inequalities are listed below.

- Odd antihole inequality (Nemhauser and Trotter, 1973): if $\overline{C}$ is an odd antihole (i.e., complement of an odd hole), then

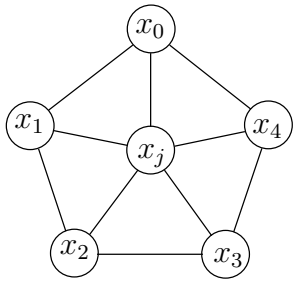
  $$\sum_{i \in \overline{C}} x_i \leq 2$$

  is a valid inequality to (SP).



- Wheel inequality (Grötschel et al., 1988): a wheel is an odd cycle C with an additional node j, called *rim*, connecting to all nodes of the cycle. The wheel inequality associated with a wheel (C, j) is

$$\frac{|C|-1}{2}x_j + \sum_{i \in C} x_i \leq \frac{|C|-1}{2}.$$



- Antiweb and web inequalities (Trotter, 1975): a web is a graph $W_{(n,k)} = (N, E)$ ($|N| = n \geq 3$) in which

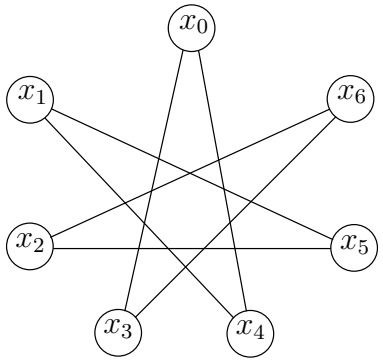
$$(i, j) \in E \Leftrightarrow j = i + k, i + k + 1, \ldots, i + n - k,$$

where all sums are taken modulo n. If $W_{(n,k)}$ is a web, we have a valid inequality

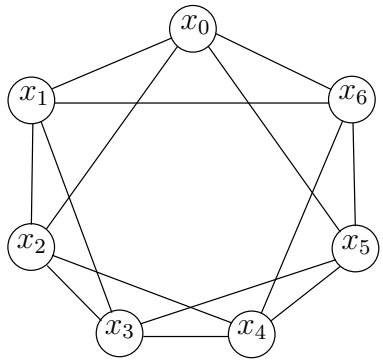
$$\sum_{i \in N} x_i \leq k.$$

The complement $\overline{W}_{(n,k)}$ of a web $W_{(n,k)}$, called antiweb, also contributes a valid inequality

$$\sum_{i \in N} x_i \leq \lfloor n/k \rfloor$$



$W_{(7,3)}$



$\overline{W}_{(7,3)}$

In this section, two important valid inequalities are reviewed: the clique inequality and the odd cycle inequality. They will be implemented in our code. One of the reasons to use them is that all left hand side coefficients of them are zero-one. This will guarantee that the storage structures for inequalities are coded easily and efficiently in space and in speed as well. The generalization of clique and odd circle inequalities relates to the work of Chvátal (1975). In one of his theorems, there is a valid inequality

$$\sum_{j \in N} x_i \leq \alpha(G),$$

where $\alpha(G)$ is the maximum cardinality of stable sets of $G = (N, E)$.

The reader is referred to Borndörfer (1998) for a good survey of the valid inequalities of set packing polytope.

3.2.2 Set covering facets

Besides trivial facets, such as $x_j \leq 0, j \in N$ (if and only if $|\mathcal{I}_i \setminus \{j\}| \geq 2$ for each $i \in M$), $x_j \leq 1, j \in N$, Sassano (1989) discovers several facets with all left hand side coefficients being zero. The idea starts from the following definition:

Definition 3.2.3. Let $G = (M, N, E)$ be a bipartite graph associated with a zero-one matrix A in which M is the row index set, N is the column index set, and the edge $(i, j) \in E : i \in M, j \in N$ if $A_{ij} = 1$. Let $H = (M_H, N_H, E_H)$ be a subgraph of G, then we call

$$\sum_{j \in N_H} x_j \geq \beta(H)$$

rank inequality associated with H. $\beta(H)$ is the minimum cardinality of a cover in H and also called *covering number* of H. In a similar way, a facet defined by a rank inequality is called *rank facet*.

We will introduce 2 facets discovered by Sassano which are contained in a class of rank facets.

- Let $R(p, t, q) = (M_R, N_R, E_R)$ be a (q, t)-rose of order p. We have the following facet of the associated matrix of R

  $$\sum_{j \in N_R} x_j \geq \left\lceil \frac{(t - q + 1)p}{t} \right\rceil$$

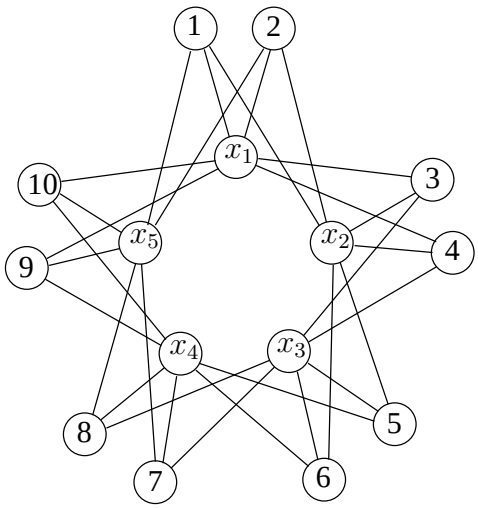
  if and only if $p(q - 1) \neq 0 \pmod{t}$, or $t = p$.

- Let $R(p, q, q) = (M_R, N_R, E_R)$ be a q-rose of order p, $p \neq q$. We have a following facet of the associated matrix of R

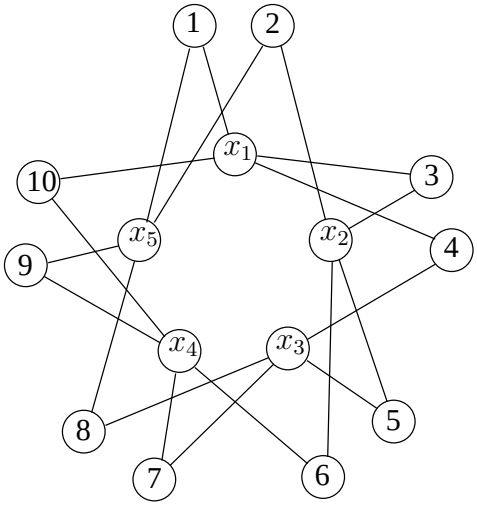
  $$\sum_{j \in N_R} x_j \geq \left\lceil \frac{p}{q} \right\rceil$$

  if and only if $p \neq 0 \pmod{q}$.

The proof of these facets and associated definitions can be found in Sassano (1989). In the same paper, Sassano also introduces the method to lift the facet to be globally valid. The figure below gives an example of a 3-rose of order 5.



3-rose of order 5



(2,3)-rose of order 5

The thesis of Borndörfer (1998) reviews more facets of the set covering polyhedron. It also introduces necessary and sufficient conditions for a rank inequality to become a facet which was shown by Euler et al. (1987) and generalized by Laurent (1989).

3.3 Computational Issues

3.3.1 Data structure

Section 3.1.2 claims the sparsity of the constraint matrix of a (SPP) generated from a crew pairing problem. Generally, it is also a property of set partitioning matrices modelling scheduling problems which are dealt by the two-step method mentioned in Section 3.1. The column *nnz* of Table 3.1 demonstrates the number of nonzeros in the constraint matrix of most difficult test problems which are obtained from a well-known test set mentioned in Hoffman and Padberg (1993), Borndörfer (1998), from Vietnam Airlines problems discussed in Section 2.4, and from randomly generated problems which will be described later. All problems in the first set have less than 9 nonzeros in each column and those of the second set have less than 6 in each column. This sparsity, with zero-one coefficients, needs a special data structure in order to get efficiency.

A widely-used structure to store such sparse zero-one matrices is *column major format* (or *row major format*). It only stores row indices of nonzero coefficients. In addition, this storage method uses two other arrays: one to store the number of nonzero entries in a column and another to store the index to the major array where the row index of the first nonzero of the column located. It is not convenient to scan a matrix stored in the column major format by row oriented operations. In such a case, we can additionally store the matrix using the row major format. Certainly, two these data structures must be synchronized to be the same. Besides saving memory, another advantage of this storage method is to load the matrix quite quickly into a linear solver

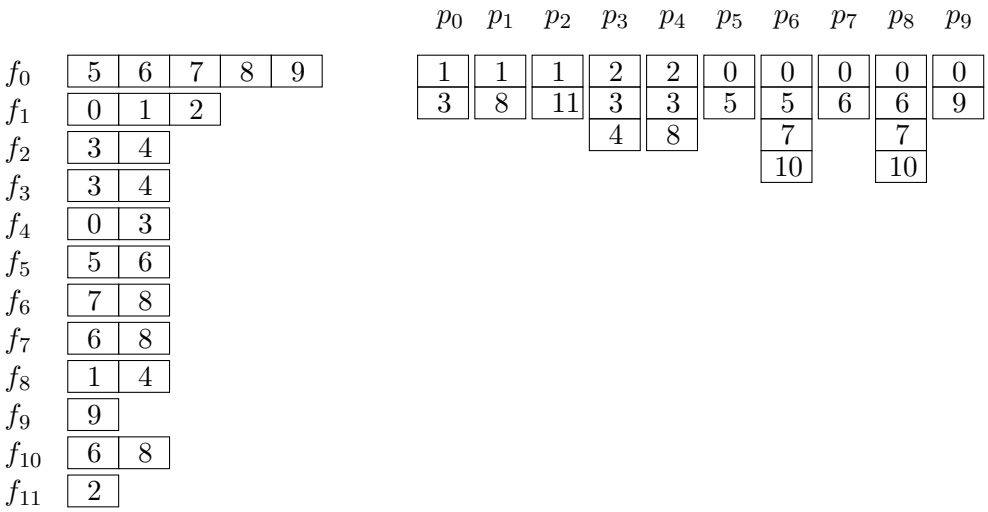


Figure 3.1: The zero-one matrix of the example shown in Section 3.1.2 is stored in a set of sparse format rows and in a set of sparse format columns

(e.g., CPLEX) which supports the sparse matrix storage format. Details of the formats are found in CPLEX (1997), Hoffman and Padberg (1993), Borndörfer (1998).

The thesis solves the storage problem in a different way. Every column (row) is kept in a *separate* array which stores only indices of nonzero elements. This means that the whole matrix is not considered as an entity. Although a set of the columns is enough to present the matrix, a different set is needed to hold the rows (also in the sparse format). The storage method needs an equivalent space for a matrix in comparison with the column (row) major format mentioned in the paragraph above. The required space depends mainly on the number of nonzero coefficients. The separately stored columns can require more extra space for their arrays to be extendable. An example of this storage method is depicted in Figure 3.1.

Additionally, there are the following reasons to use the format.

- Row oriented and column oriented operations are performed easily on the data structures.
- ABACUS (Thienel, 1995) is used as a framework for the implemented set partitioning solver. The usage of this library requires users to define a *separate* data structure for each kind of constraint (variable). This also means that a data structure for a whole constraint matrix (as done by other people) is not acceptable to work with ABACUS. A brief discussion of ABACUS will be introduced in Chapter 5. More internal design of the library can be found in Thienel (1997).
- Data objects for constraints and variables will be exchanged individually within a parallel computing environment. We do not have to extract some rows (columns) out of the matrix before sending them. This also means reducing the computation time. We will go back to this topic later in Chapter 5.

Every variable (and constraint) is assigned a *global* identification (ID). Since the constraints of the original integer formulation are global, they should be implemented to remember which variables they cover (i.e., which columns the corresponding coefficients are 1).

The transformation of constraint matrices from user data structures to structures used by linear solvers is done by ABACUS. Users are required to supply functions to determine the value at a position on the matrix. These functions could be quite slow if we do not consider their design carefully. The simplest function, named `coeff()`, is to determine the value at a point on the matrix. Since the list of IDs which is covered by a row is sorted according to the cardinality of IDs, checking the existence of a variable ID on the variable ID list of the row is best performed by a binary search.

Although a binary search has been involved, it is still easily seen that the process of regenerating a whole row by iteratively calling the function `coeff()` is time consuming. Therefore, ABACUS supplies a function `genRow()` which can be overridden by users to regenerate the whole row at a time (i.e., ABACUS does not determine point-by-point and gather them to create the row). The input of the function is a list V of column IDs whose coefficients on the row are required.

Figure 3.2: Nonzero coefficients of the constraint matrix of the problem sppnw32

Note that Section 3.1.2 comments that there are a large number of variables and that rows are very sparse. Figure 3.2 visualizes the constraint matrix of a typical (SPP) generated from a crew pairing problem. Therefore, instead of traversing one-by-one through V, it should be better to bypass a given number s of variables before restarting the next search. This means that if we go ahead s variables and see that the variable ID at that position is less than the current variable ID of the row, we will jump to that point immediately. Otherwise, we make a search on these variables.

The idea of the new approach is sketched in Function 3.3.1. This technique also has an advantage of utilizing the fact that many nonzero coefficients often stay nearby. With the function, the LP loading time is reduced significantly. The constant s can be set depending on the sparsity of the row. The worst case complexity of the technique is easily seen to be $O(|N_i| + |V|)$ if $s = 1$.

3.3.2 Preprocessing

In this section, we focus on how to remove constraints and variables which are *redundant* in the integer model. In general, the main purpose of preprocessing techniques is to reduce the size of coefficients in the constraint matrix and the size of bounds on the variables as well. In our case, all coefficients of the constraint matrix are zero-one. Therefore, special techniques of preprocessing the problem are involved. A good discussion on preprocessing and probing techniques for mixed integer programming

Function 3.3.1 `genRow`(N^i, V)

```
k ← 0, l ← 0
repeat
  k_next ← k + s
  if V_{k_next−1} < N^i_l then
    k ← k_next
  else
    while (k < k_next) ∧ (l < |N^i|) do
      if V_k < N^i_l then
        k ← k + 1
      else if V_k > N^i_l then
        l ← l + 1
      else
        A_{iV_k} ← 1 {only nonzero coefficients are needed}
        k ← k + 1
        l ← l + 1
      end if
    end while
  end if
until (k ≥ |V|) ∨ (l ≥ |N^i|)
```

problems is exhibited by Savelsbergh (1994). As for the problems of zero-one constraint matrices, Hoffman and Padberg (1991) present techniques to deal with their special property. Borndörfer (1998) reviews many good preprocessing techniques for the set partitioning problem. Although there are many preprocessing and probing techniques, some of them have been chosen due to their efficiency on tightening the set partitioning problems generated from crew pairing problems.

Duplicated columns (P1)

As commented in Section 3.1.2, a given timetable is duplicated to several consecutive periods. This increases the ability that a group of flights could be covered (or presented) by different pairings as shown in Figure 3.3. Costs of two pairings in the figure are possibly different and we only need to keep one of them in the model. Consider two columns $k, l \in N$. Generally, it is quite clear that if $M^k = M^l$ and $c_k \geq c_l$, the column k can be removed without any change to the problem.

Determining all duplicated columns is a computational problem. Denote the maximum number of nonzeros of a column by cnz. We assume that all row and column identifications stored in columns and rows are sorted. Another easy-to-obtain assumption is that columns are ordered lexicographically which can take the time of $O(n \times \ln(n) \times cnz)$ to be done. The worst case complexity to find all duplicated columns is $O(n \times cnz)$.

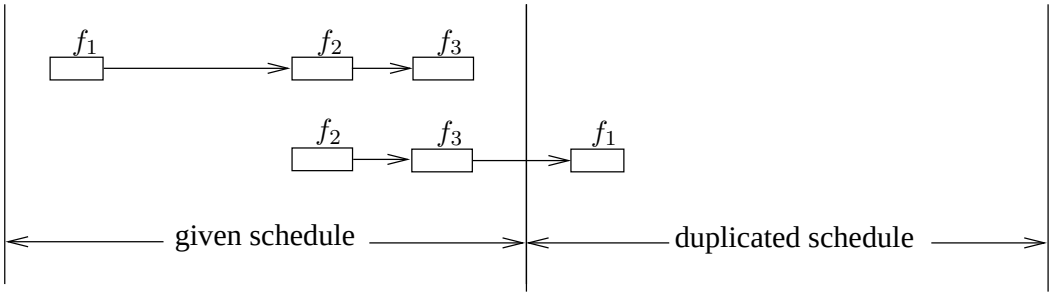


Figure 3.3: Two pairings cover the same group of flights, creating two duplicated columns in the set partitioning model

Dominated rows (P2)

Figure 3.4 demonstrates an example of the domination of rows. Both flights f_1, f_2 are elements of pairings p_1, p_2 and p_3. However, f_2 is also covered by the additional pairing p_4. Since at least one of the pairings covering f_1 belongs to a feasible solution, p_4 becomes redundant. We say that the constraint for f_1 is *dominated* by that of f_2 which can be removed. The idea of row domination becomes clearer when we look into two corresponding constraints below:

$$\begin{aligned} f_1: \quad & x_1 + x_2 + x_3 & = 1 \\ f_2: \quad & x_1 + x_2 + x_3 + x_4 & = 1 \end{aligned}$$

Variable x_4 of the second constraint can be set to zero. More generally, for two rows $k, l \in M$, if $N^k \subseteq N^l$, we can set all $\{x_j : j \in N^l \setminus N^k\}$ to 0 and remove the row l.

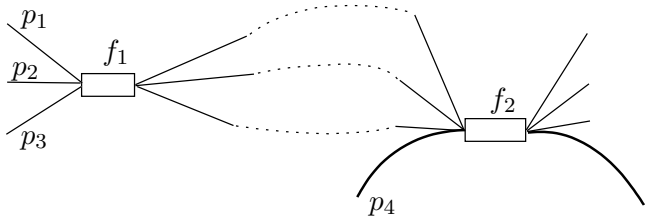


Figure 3.4: Two flights are covered by the same group of pairings. The second flight is covered by an additional pairing p_4

In order to find dominated rows, the method discussed in Borndörfer (1998) will be used. The procedure comes from the the following idea: the set of rows which dominates the row i is equal to

$$(\bigcap_{k:i\in\mathcal{J}^k} \mathcal{J}^k) \cap M.$$

The technique could be understood as a process of iteratively removing rows which do not have the same set of pairings to the considered row. The worst case complexity of determining all dominated rows is $O(m \times rnz \times cnz)$. Note that rnz denotes the maximum number of nonzero coefficients on a row.

Row cliques (P3)

This preprocessing technique is mainly based on the set partitioning model. If a column k is neighbors to every column l of a row, x_k can be removed. (Two columns k, l are neighbor to each other if $M^k \cap M^l \neq \emptyset$.) The idea behind is quite simple: there is at least one column of the row equal to 1 in a feasible solution. Thus, x_k must be 0. Given a column k, we can easily compute the set of its neighbor columns. The set is $\bigcup_{l \in M^k} N^l$. From this computation, we can determine columns which are neighbors to every column of a row by intersecting neighbor sets of all columns of the row. It is easily seen that the complexity of the mentioned procedure is $O(rnz^2 \times cnz)$.

Row singleton (P4)

If a row has only one nonzero coefficient, the associated variable can be set to one. Furthermore, other variables on rows which contain this variable can be removed from the model. The time to compute all singleton rows and remove variables is $O(m \times cnz)$.

The preprocessing techniques also become useful for non-root problems of a branch-and-bound tree. A preprocessing technique could be recalled after other techniques have processed the constraint matrix. For example, if we remove some constraints, two different columns can turn out to be the same and one could be removed due to the column duplication. Therefore, the preprocessing will be put in front of the cutting plane stage in the implementation.

An effective post processing technique, called *reduced cost fixing* should be reminded. Denote by z^* the optimal value of the current linear relaxation, $\overline{z}$ the best known feasible integer solution. If $x_j^* = 0$ and $\overline{z} \leq z^* + \overline{c}^j$, variable j can be eliminated. Fortunately, the reduced cost fixing has been implemented into ABACUS.

3.3.3 Branch-and-bound

In solving a combinatorial optimization problem by branch-and-bound, there are several algorithmic choices which have a strong impact on the performance of computation. Some of the most important choices are:

- Branching strategy,
- Node selection strategy.

In this thesis, a thorough study of how to branch a branch-and-bound node (node, for short) and how to select the next open node will not be given. Many basic points in this are addressed in the textbooks of Nemhauser and Wolsey (1988), Wolsey (1998). One of the best surveys on search strategies is possibly the one by Linderoth and Savelsbergh (1999).

Branching Strategy

Linderoth and Savelsbergh (1999) classify branching strategies into two general types: *variable dichotomy* and *GUB dichotomy* (GUB stands for generalized upper bound). The variable dichotomy divides a node into two (possibly more) child nodes by imposing new bounds on a variable. For example, given a variable x_j with a fractional value x_j^*, we can branch the current node into a left node with an additional constraint $x_j \leq \lfloor x_j^* \rfloor$ and a right node with an additional constraint $x_j \geq \lceil x_j^* \rceil$. Certainly, constraints of the father node are also included in the child nodes. In a different way, the GUB dichotomy pays attention to GUB constraints of the form $\sum_{j \in T} x_j = 1$ (or $\sum_{j \in T} x_j \leq 1$), where T is a subset of the column index set. If there exists a subset T' of T satisfying that $0 < \sum_{j \in T'} x_j < 1$, the GUB dichotomy uses the constraint $\sum_{j \in T \setminus T'} x_j = 0$ for the left node and the constraint $\sum_{j \in T'} x_j = 0$ for the right node.

Although ABACUS has implemented *strong branching*, it still requires the support from linear solvers. It will be used in computation because it is quite useful for degenerate problems like set partitioning problems.

Some additional methods for choosing a variable to be branched have been implemented. Let F be the index set of fractional variables. The implementation has the following variable branching schemes.

- `ValueTimeObjectiveCoefficient`: choose $j = \arg\max_{j \in F}\{c_j \min\{x_j^*, 1 - x_j^*\}\}$.
- `PadbergRinaldi` (Padberg and Rinaldi, 1991): this method is quite complicated, intending to obtain two targets simultaneously: x_j^* is close to $1/2$ and $|c_j|$ is large. In order to find j, the method firstly computes two values:

  $$L = \max_{j \in F}\{x_j^* : x_j^* \leq 1/2\} \qquad U = \min_{j \in F}\{x_j^* : x_j^* \geq 1/2\}.$$

  Now, with a given parameter $\alpha \in [0, 1]$ (by default, $\alpha = 0.25$), choose $j = \arg\max_{j \in F}\{|c_j| : (1 - \alpha)L \leq x_j^* \leq U + \alpha(1 - U)\}$.
- `StrongEffect`: choose $j = \arg\max_{j \in F}\{|N(j)|\}$. (Note that we denote by $N(j)$ the set of neighboring variables of j in the associated intersection graph.)

A GUB branching is also implemented in the branch-and-cut code. Since every set partitioning constraint can be a candidate for GUB branching, one of them should be chosen. In this case, the constraint with the greatest number of fractional variables is selected. The most important advantage of the GUB dichotomy is that it is believed to create a more balanced branch-and-bound tree. The reason is that the branching creates two similar child problems. The list of the fractional variables is sorted according to an increasing order of their relaxation values into $x_{j_1}^*, x_{j_2}^*, \ldots, x_{j_{|T|}}^*$. (Therefore GUB branching is also called special ordered set (SOS) branching.) After that, we build a set $T' = \{j_{2k-1} : 1 \leq k \leq \frac{|T|+1}{2}\}$.

Another branching strategy which looks similar to GUB dichotomy is also investigated. It is based on an idea of Ryan and Foster (1981).

Theorem 3.3.1. *If A is a zero-one matrix, and a basic solution x^* to $Ax = 1$ is fractional (i.e., at least one component of x^* is fractional), then there exist two rows i' and i'' such that*

$$0 < \sum_{j:A_{i'j}=1, A_{i''j}=1} x_j^* < 1 \tag{3.11}$$

Consider a fractional variable $x_{j'}$ and a row i' containing $x_{j'}$ (i.e., $A_{i'j'} = 1$). Since $A_{i'.}^\top x^* = 1$, there exists at least another basic column $x_{j''}$ which $x_{j''}^*$ is fractional. It is quite clear that two basic columns cannot be the same. Therefore, we can easily find another row i'' such that $A_{i''j'} = 1$ or $A_{i''j''} = 1$, but not both. We have

$$\begin{aligned} 1 &= A_{i'.}^\top x^* \\ &= \textstyle\sum_{j:A_{i'j}=1} x_j^* \\ &> \textstyle\sum_{j:A_{i'j}=1, A_{i''j}=1} x_j^* \end{aligned}$$

because only one of $x_{j'}^*$ and $x_{j''}^*$ is included in the last summation. The left inequality of (3.11) is obvious because $x_{j'}^*$ and $x_{j''}^*$ are fractional.

From the theorem above, we can create the two following constraints, one for each disjoint subregion:

$$\sum_{j:A_{i'j}=1, A_{i''j}=1} x_j = 0 \qquad \sum_{j:A_{i'j}=1, A_{i''j}=1} x_j = 1.$$

The meaning of the Ryan-Foster branching is clearer in the context of branch-and-price where the method is widely used. We will go back to its details in Chapter 4.

Node Selection Strategy

A good node selection strategy should deal effectively with two goals: finding a good feasible integer solution and proving that no solution is better than the current solution. Linderoth and Savelsbergh (1999) divide node selection strategies into four categories: *static*, *estimated-based*, *two-phase*, and *backtracking*.

The thesis does not go into details of enumeration strategies. A new strategy is added to the implementation. It is called *least-fractional variable first* which means to pick the node containing the least number of fractional variables first for computation. Node selection strategies provided by ABACUS will also be used in computation. Certainly, the most popular method is *best-first search* in which the node with the smallest relaxation value of z^* will be explored first. In this manner, the method tries to improve the global lower bound, and hence, possibly reduces the total number of branch-and-bound nodes.

3.3.4 Cutting plane generation

Section 3.2.1 shows important discovered facets of a set partitioning polytope. But in the implementation, only clique and odd cycle inequalities are generated. In this section, we will go to the problem of how to generate them and lift zero-valued variables

to make them stronger. Remember that all left hand side coefficients of these valid inequalities are zero-one, similar to the constraints of the original formulation. Another point which should be remembered is that the inequalities used in the implementation are globally valid. Although general cutting planes, such as the Chvátal-Gomory cutting planes, the lift-and-project cutting planes, have been revisited and introduced good performance, they are locally valid. This can lead to a memory problem when solving very large scale crew pairing problems. One good paper describing how to use lift-and-project cutting plane in solving zero-one problems is written by Balas et al. (1993). In another paper, Balas et al. (1996) also present a way to lift Gomory cuts to be globally valid.

Clique Inequalities

As shown before, clique inequality is a facet of the polytope. However, separating clique inequalities is a difficult problem (Grötschel et al., 1988). Therefore, only fast methods of generating the same style of inequalities on complete subgraph will be discussed.

- *Row-lifting* (Hoffman and Padberg, 1993): The main idea of the method is starting from a small complete subgraph which can be obtained easily from the constraints of the model. Since we want to cut off the current linear relaxation point x^*, only fractional variables of the constraint will be used for the starting set. Then, we try to find other fractional variables into this set. Remember that the number of fractional variables is not very large with respect to the total number of variables. Finally, zero-valued variables will be lifted into the set of variables. Since there are a lot of zero-valued variables, we only choose randomly a small set of them to be lifted. In computation, the number of randomly chosen variables is determined based on the difference between the number of rows and the number of columns of the constraint matrix. Obviously, unless there is no fractional variable lifted into the set, the final inequality violates x^*.

- *Greedy heuristic* (Borndörfer, 1998): In this method, the starting variable set of the valid inequality comprises only one fractional variable. The method will then try to include other fractional variables into the set. Similar to the row-lifting, zero-valued variables will be included into the set. However, they are computed in a different way. We find all variables neighbor to the fractional set. In order to guarantee that they create a complete subgraph, the set of all these zero-valued variables will be intersected with the rows to find out the maximum cardinality set. Finally, they will be inserted into the fractional set to make the final inequality. As remarked by Borndörfer, the fractional variables which will be chosen as a seed of the clique is sorted according to their values. This greedy method does not guarantee the final fractional clique violates x^*. Therefore, an additional check is needed.

- *Recursive Smallest First* (Carraghan and Pardalos, 1990): This method aims at solving the following recursive equation:

$$\max_{Q \text{ clique in } G} \sum_{k \in Q} x_k^* = \max\{\ x_j^* + \max_{Q \text{ clique in } G[N(j)]} \sum_{k \in Q} x_k^*, \quad \max_{Q \text{ clique in } G-j} \sum_{k \in Q} x_k^*\}, \tag{3.12}$$

where $G[N(j)]$ is the graph induced from the node set $N(j)$. The efficiency of solving the equation depends much on the way of choosing the variable j. One is to choose the variable which has the smallest number of incident edges in the associated intersection graph. Since it can take an exponential number of steps to solve 3.12, the motivation of such a choice is to reduce the number of steps. We must have another mechanism to prevent the algorithm to run very long. The implementation has two additional parameters α and β. If the number of vertices of the intersection graph is less than α, a complete enumeration will be performed. Otherwise, we turn to control the depth of the recursion by β. Note that we choose the depth first search strategy in solving the recursive equation. If the results of the left and right branches are the same, we should choose the clique having a larger number of vertices.

Odd Cycle Inequalities

The inequalities can be separated in polynomial time by an so-called GLS algorithm (Grötschel et al., 1988). The main idea of the algorithm is to transform the separation problem to finding the shortest path on a new graph which is generated from the intersection graph of the fractional variables. Let $G = (V, E)$ be this intersection graph. We construct the new bipartite graph G_B as follows: the node set of G_B consists of two copies V' and V'' of V''; an edge $u'v''$ is in G_B if uv is in G.

We easily realize that a path P_u from u' to u'' in G_B corresponds to an odd cycle C_u in G. The weight $w_{u'v''}$ is assigned with $1 - x_u^* - x_v^*$. By doing so, we have

$$\begin{aligned} w(C_u) &= \textstyle\sum_{u'v'' \in P_u}(1 - x_u^* - x_v^*) \\ &= |C_u| - 2\textstyle\sum_{u \in C_u} x^* u. \end{aligned}$$

Therefore, we have the following equivalences:

$$w(C_u) < 1 \Longleftrightarrow |C_u| - 2\sum_{u \in C_u} x_u^* < 1 \Longleftrightarrow \sum_{u \in C_u} x_u^* > \frac{|C_u| - 1}{2}.$$

This proves that an odd cycle having the weight less than 1 corresponds to a violated odd cycle inequality. The most violated odd cycle can be found by solving the shortest path problem on the associated graph G_B. In the implementation of the algorithm, when labelling the neighboring nodes, it is only necessary to consider labels whose distances are less than 1. Note that, with the weight assignment above, $0 < w_{u'v''} < 1$.

3.3.5 Primal heuristics

One of the difficult problems associated with the branch-and-bound approach is that the number of nodes grows significantly. This leads not only to the memory starvation, but also to a rather time consuming computation. In such a case, a feasible solution is quite important to fathom nodes which cannot give a better solution. In addition, a good integer solution can be used to fix and set variables using the reduced cost fixing mentioned in Section 3.3.2. In the implementation, the information of linear relaxations in the framework of branch-and-cut is reprocessed further in order to find feasible solutions. Two LP-based primal heuristics will be discussed in this section and used in the branch-and-cut code.

Dive-and-fix

The idea of this method is to solve the linear relaxation of an integer problem and fix some fractional variables to suitable bounds. Certainly, if we fix variables which are nearer to integer points, there are more possibilities that the remaining problem is integer feasible. The method is more precisely stated in Algorithm 3.3.2.

Algorithm 3.3.2 Dive-and-fix heuristic

1: $i \leftarrow 1$
2: **repeat**
3: solve the problem (LP) $z^* := \min\{c^\top x : Ax = 1, x \geq 0\}$ to obtain the solution x^*.
4: **if** ((LP) is infeasible or unbounded) $\vee$ $(z^* \geq \overline{z})$ **then**
5: **break**
6: **else**
7: sort $F = \{x_j^* : 0 < x_j^* < 1\}$ in ascending order of $\min\{x_j^*, 1 - x_j^*\}$.
8: fix first α variables of F to their nearest integer points.
9: **end if**
10: $i \leftarrow i + 1$.
11: **until** $i > \beta$
12: **if** (x^* are integer) $\wedge$ $(z^* < \overline{z})$ **then**
13: $\overline{z} \leftarrow z^*$.
14: **end if**

Since the algorithm can take quite long time for large problems, the algorithm is permitted to run for β loops. The dive-and-fix heuristic has many variants. A general discussion of the method can be found in Wolsey (1998).

Near-integer-fix

It can be said that the second heuristic is also a variant of the dive-and-fix heuristic. Instead of simply fixing fractional variables nearest to integer points, the second method

employs a more complicated technique of choosing variables to be fixed. A main difference between these two variants is that the near-integer-fix heuristic will not fix a given number α of variables. It will fix all fractional variables whose integer distances fall below a given number. The method is presented in Algorithm 3.3.3.

Algorithm 3.3.3 Near-integer-fix heuristic

1: $i \leftarrow 1$
2: **repeat**
3: solve the problem (LP) $z^* := \min\{c^\top x : Ax = 1, x \geq 0\}$ to obtain the solution x^*.
4: **if** ((LP) is infeasible or unbounded) $\vee$ $(z^* \geq \overline{z})$ **then**
5: **break**
6: **else**
7: find $x_{\min} := \min\{x_j^* : x_j^* \geq \gamma\}$.
8: find $x_{\max} := \max\{x_j^* : x_j^* \leq 1 - \gamma\}$.
9: **if** $|x_{\min} - x_{\max}| \geq \gamma$ **then**
10: fix $\{x_j^* : x_j^* < x_{\min}\}$ to 0.
11: fix $\{x_j^* : x_j^* > x_{\max}\}$ to 1.
12: **else**
13: fix $\{x_j^* : x_{\min} \leq x_j^* \leq x_{\max}\}$ to 0.
14: **end if**
15: **end if**
16: **until** $i > \beta$
17: **if** (x^* are integer) $\wedge$ $(z^* < \overline{z})$ **then**
18: $\overline{z} \leftarrow z^*$.
19: **end if**

Another change in the near-integer-fix belongs to lines 12 and 13 of the algorithm. When the segment $[x_{\min}, x_{\max}]$ is extremely small, all fractional variables which lie within this segment will be fixed to 0. The method is quite similar to a heuristic used by Bixby et al. (1995). The main difference between them is that the suggested algorithm will try to fix all variables falling between $[x_{\min}, x_{\max}]$ (Bixby et al. fix a variable having the smallest index to 1). By that way of fixing, it is easier to obtain a feasible solution because fixing a variable to 1 will have a strong propagation on other variables.

There are still many heuristics suggested for solving set partitioning problems. Some of them are: dual heuristics of Fisher and Kedia (1990), the dual cost perturbation heuristic of Wedelin (1995), small set partitioning heuristic mentioned by Linderoth et al. (2001). More discussions on heuristics used for set covering/partitioning problems could be found in Nemhauser and Wolsey (1988).

3.3.6 Computational results

The computing environment is set up as following. A computer with an Intel Pentium III (Coppermine) 800 MHz processor is supplied with a memory of 500 MB. In addition to the real memory, the computer is also supported with a swap space of 1 GB hard disk. The system runs a Linux operating system which is tuned to perform computational tasks only. This section will show the computational results according to the variants of algorithmic implementation and parameters.

The most favorite compiler on Linux systems is GCC, which is a GNU compiler. This compiler is widely used in the open source community. Although it is not optimized well to a specific processor family, it now supports many computing platforms and is being enhanced to generate better executable codes. The branch-and-cut code is compiled by the compiler of version 2.95 with an optimization flag "`-O2`". The loop unrolling is disabled because it possibly annoys the computational performance by unclear effects. With the support of ABACUS, the code is implemented easily because the skeleton of a branch-and-cut code has been done by this framework. The linear solver in our experiment is CLP of the project COIN by IBM (more information can be found on its web site). This LP solver supports a steepest edge pricing in two simplex methods. It also provides a barrier method with cross-over. However, all other linear solvers supported by ABACUS could be used as well. CLP is configured with the *full* steepest edge pricing. Most researchers in operations research are quite acquainted with CPLEX, which is a strong commercial solver for linear relaxation. Nevertheless, since we would like to parallelize the set partitioning solver, a free linear programming solver is preferred to CPLEX, which is too expensive to buy a license for a large number of computers.

We will revisit the computational issues mentioned in the previous sub-sections in context of solving set partitioning problems which are generated from crew pairing problems. Obviously, since the thesis does not have enough space to present the computational results of many combinations of the algorithmic and parametric aspects, only some of the typical ones are taken into account in order to to show important points of the theory and the implementation.

Three sets of test problems as shown in Table 3.1 are used in computation. The first set comes from the well-known set published by Hoffman and Padberg (1993), and used by many researchers to test their algorithms and implementations. They generated this set from airline crew scheduling problems modelled as set partitioning problems. Interestingly, many of them can be solved without branching or even cutting plane generation phase. Since the thesis does not have enough space to present all problems, we will only focus on some of the most difficult problems. More discussions can be found in Hoffman and Padberg (1993), Borndörfer (1998).

The second set is produced from timetables of Vietnam Airlines which were introduced in Section 2.4. The schedule table for the second set is a domestic schedule which was operated in April 20, 2004. The original problems are transformed to set partitioning problems by the method presented in Section 3.1.2. There should be a remark on

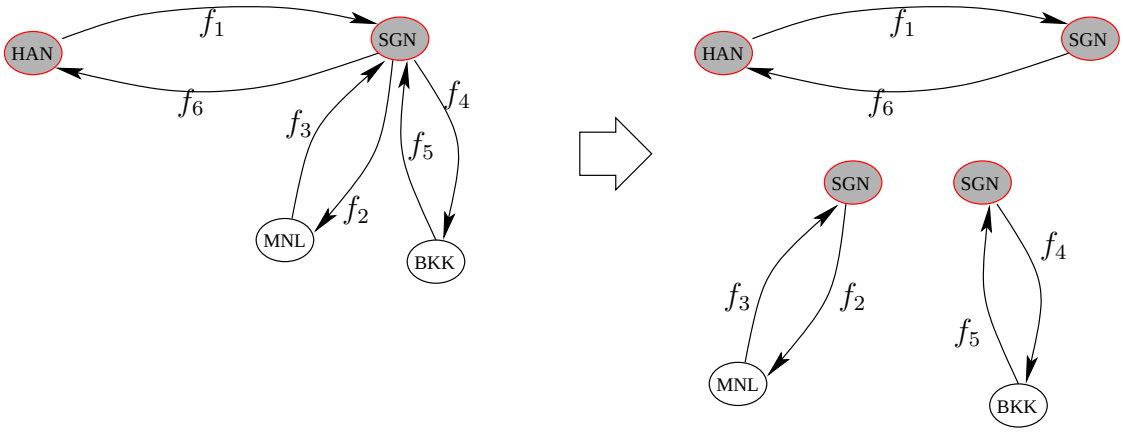


Figure 3.5: Decomposing a pairing into 3 sub-pairings

the difference between "vn320r" and "vn320". Because of the special structure of the flight network of Airbus 320 (see Figure 2.3(b)), a huge number of generated pairings can overload the memory of testing computers. Most of them have the form depicted in Figure 3.5. Also shown in the figure, we can see that the pairing can be decomposed into 3 feasible sub-pairings whose total cost is worse than the original (because of the cost structure of Vietnam Airlines). If the original belongs to a feasible solution, the new solution which replaces it with the three sub-pairings is also feasible. Furthermore, pairings like the three sub-pairings together can replace a lot of longer pairings. Therefore, instead of generating all feasible pairings, only non-decomposable sub-pairings are generated. After having an optimal solution, we can improve the objective cost by merging sub-pairings. The problem "vn320r" will be used in experiments. The generation of such a reduced problem consumes more time to decompose the long pairings and check the existence of their sub-pairings in the current set of pairings. Therefore, do not be surprised with the higher enumeration time associated with "vn320r".

The third set is generated randomly from a network visualized in Figure 3.6. Basing on the current network of the aircraft ATR 72, some connections are added between the subsidiary airports which are likely to be opened in future. This will create more possibilities of next flights for a crew. In order to create more realistic problems, the data is generated from flight routes, which are paths that aircrafts will take during a scheduling horizon (for example, day schedule, week schedule). This seems more reasonable than creating flights randomly in the scheduling period without paying attention to the flight routes. Observing the activities of Vietnam Airlines, the flight routes of aircrafts are classified into 2 types: daily route and weekly route. If an aircraft performs a daily route, it departs from a base and returns to that base the same day. Moreover, that route is repeated almost every day of a week. Carrying out a weekly route, an aircraft could stay far a way from its base for several days. This often happens with long haul flights. In order to guarantee the reality of the test problems, two types of routes are involved in which the proportion of the number of daily routes to the number of weekly routes in schedules of Vietnam Airlines is kept unchanged.

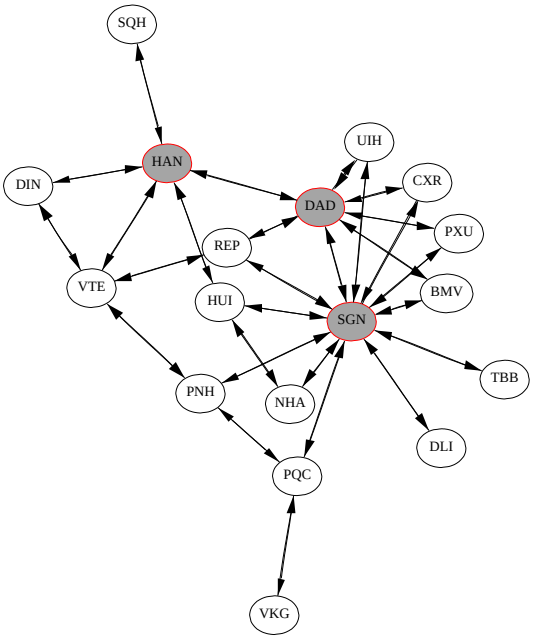


Figure 3.6: The flight network for randomly generated flight sets

Four timetables generated from that flight network have a prefix "vncpp", containing 258 flight legs each. Due to the rules and regulations of the airline, there are very many feasible pairings. Similar to the way of choosing problems in the first test set, only those which give fractional solutions to first linear relaxation are chosen.

Name	m	n	nnz	t_{enu}	preprocessing					
					n^-_{P1}	m^-_{P2}	n^-_{P2}	n^-_{P3}	m^-_{P4}	n^-_{P4}
aa01	823	8904	72965	-	0	206	1217	143	0	0
aa04	426	7195	52121	-	0	83	995	77	0	0
kl02	71	36699	212536	-	20157	0	0	0	0	0
nw04	36	87482	636666	-	41292	0	0	0	0	0
us01	145	1053137	13636541	-	682495	59	19624	0	0	0
*vn320	297	1755057	10137809	0:08:05	7550	0	0	0	0	0
vn320r	297	334407	1796874	0:30:01	7550	0	0	0	0	0
vn321	168	22090	121379	0:00:24	129	0	0	0	0	0
vnAT7	296	739428	4329537	0:09:36	8259	0	0	0	0	0
vncpp1	258	116533	645918	0:00:47	22829	0	0	0	0	0
vncpp2	258	110283	611870	0:00:46	19390	0	0	0	1	1
vncpp3	258	101281	560585	0:00:42	21255	0	0	0	0	0
vncpp4	258	87678	483001	0:00:37	15094	0	0	0	0	0

Table 3.1: Set partitioning test problems solved by the branch-and-cut code

Preprocessing

The preprocessing techniques only consider the set partitioning model and detect variables and constraints to be removed. However, some of them are basic variables or nonbasic constraints which should not be removed from the model. This means that only nonbasic variables which have been detected to be removable will actually be

removed. For constraints, only basic ones will leave the model. The motivation behind this approach is that we would like to keep the feasible basis of the last linear relaxation. However, for the root node, the preprocessing techniques are performed repeatedly until there are no variables or columns for removal. It is quite clear that after being preprocessed by one technique, the matrix is possibly reduced more by other techniques. However, we can only do that with the root node (i.e., before using ABACUS). ABACUS does not delete constraints and variables right after the functions `ABA_SUB::removeCons()` and `ABA_SUB::removeVars()` have been called. Instead, it performs the removals at the beginning of the next iteration of the cutting plane algorithm. Therefore, the further preprocessing is too complicated to implement. With the row clique technique, we only start with rows containing more than 1000 nonzero coefficients. Row singleton and column singleton are not needed to be implemented separately in user codes. ABACUS supports a function which removes fixed and set variables.

Columns "n^-_{P1}", "m^-_{P2}", "n^-_{P2}", "n^-_{P3}", "m^-_{P4}", and "n^-_{P4}" of Table 3.1 show the reduction of the constraint matrix size after the preprocessing on the root node. Note that "m^-" stands for the number of rows removed by a method, and "n^-" for the number of columns removed by a method. Their subscripts denote corresponding preprocessing techniques. For a problem with a large number of columns (e.g., "us01"), the column duplication removal is quite effective. It helps remove abouthalf the number of variables. It is also the method of choice for Vietnam Airlines and randomly generated problems. However, for such cases, the column duplication does not occur many times although there are large numbers of columns.

In Table 3.3, we will see that the preprocessing techniques are also a little helpful in reducing the size of the constraint matrix, presented by the columns "n^-" and "m^-". Although the set partitioning preprocessing techniques occupy the second largest percentage of the total time (shown in Column "$\%_{\text{Pre}}$"), they do not remove many constraints and variables. Most constraints are removed by the row domination and the reduced cost fixing. Note that variables fixed by the reduced cost fixing are not included in the table. There are many removed by this technique, and the result is that constraints which have been different turn out to be the same or dominate each other. Then, the row domination will be more effective. The constraint elimination mode of ABACUS is also turned on in order to keep the computer not running out of memory.

Branch-and-bound

Most branch-and-bound based solvers use a best first search as their default node selection strategy. It is quite easy to understand because they try to reduce the gap between global lower and upper bounds. This is also motivated by the purpose of finding the optimal solution as soon as possible. But this target is not easily satisfied with highly degenerated integer problems. This can make the best first search fall into the combinatorial trap. By contrast, a heuristic approach often tends to find a feasible (not necessarily optimal) solution as quickly as possible. Therefore, it can use

a depth first search. A new node selection called least fractional first search which aims at rapidly obtaining a feasible solution is implemented. This results in reducing the memory consumption, important for dealing with very large set partitioning problems which require long-run computations.

The least fractional first search is not a new idea. However, one of the difficult questions associated is how to employ inherent advantages of different enumeration strategies and how to use them together. It is quite clear that there is a tradeoff among them. For that purpose, the implementation considers a hybrid approach in which the least fractional first search will be turned on if the solver cannot find any feasible solution after a given number of nodes having been solved using the best first search. Certainly, we count from the last found feasible solution. Right after finding a feasible solution, we switch back to the best first search. A rising problem is that it is hard to find a solution when the incumbent is extremely near to the global optimum. If we use the least fractional first search, we can spend much time to end the computation. This is because the global lower bound is not improved so much with that search. In order to overcome this phenomenon, a simple trick is used to switch backward and forward between two search strategies. A switching is only done if a strategy fails to find a solution within a controlled number of the main branch-and-bound loop.

Name	BestFirst			LeastFractionalFirst			Hybrid		
	B&B	t_{opt}	t_{total}	B&B	t_{opt}	t_{total}	B&B	t_{opt}	t_{total}
aa01	811	1:50:46	1:55:21	685	0:15:58	1:22:08	1285	1:03:11	2:45:44
aa04	2303	3:08:12	3:10:36	3549	1:32:40	2:07:49	2097	0:54:58	2:02:26
kl02	463	0:03:38	0:10:04	1631	0:02:04	0:23:22	533	0:03:40	0:10:47
nw04	405	0:02:45	0:44:32	297	0:02:45	0:31:17	359	0:02:45	0:37:35
us01	15	0:46:05	0:51:14	17	0:44:44	0:46:56	15	0:46:04	0:51:14
vn320r	1	0:14:28	0:14:28	-	-	-	-	-	-
vn321	1	0:01:07	0:01:07	-	-	-	-	-	-
vnAT7	1	0:07:43	0:07:43	-	-	-	-	-	-
vncpp1	101	0:13:53	0:17:05	427	0:04:46	0:50:34	103	0:14:02	0:16:49
vncpp2	25	0:19:37	0:21:45	25	0:19:39	0:21:46	25	0:20:01	0:22:08
vncpp3	229	0:34:54	0:48:38	621	0:48:24	1:04:03	417	0:55:22	1:05:52
vncpp4	715	0:39:53	1:09:17	877	0:04:23	1:11:56	715	0:11:30	1:06:12

Table 3.2: Branch-and-bound enumeration strategies

In Table 3.2, column "B&B" supplies the total numbers of the branch-and-bound nodes when comparing different kinds of enumerations: best first, least fractional first and hybrid. The Vietnam Airlines problems are shown once in the table because they need only one branch-and-bound node to reach optimality. If we drop other parts (e.g., heuristics, cutting plane generation) out of the code, the least fractional first search would give a greatest value of "B&B", and the best first search a smallest value of it (as shown in row "aa04"). The most important column is "t_{opt}" which shows the elapsed time to find the optimal solution. As expected, the hybrid approach finds out the optimal solutions faster than the best first approach does for the first test set. We can see in column "t_{total}" that the hybrid approach delivers a good performance as a whole. Unfortunately, this does not often happen for the third set. More running statistics of three enumeration strategies can be found in Table 3.3.

About the branching strategies discussed before, the Padberg-Rinaldi branching strategy presents the best performance. This comes from many experimental computations under a certain number of combinations of node selection strategies, preprocessing heuristics, cutting plane generation, etc. Therefore, this branching strategy is used for all computations in this chapter. In addition, the strong branching is employed with 10 variable candidates to be tested. Although CLP supplies a strong branching function, a general approach supplied by ABACUS is preferred in the implementation. There are 50 iterations for linear relaxation by a dual simplex method. The candidates are also selected upon the given branching rule (namely, Padberg-Rinaldi).

Primal heuristics

Since two methods implemented in the branch-and-cut code are based on the dive-and-fix principle, they are most helpful in the early branch-and-bound nodes which are still easily feasible if we fix some variables to their bounds. In the near-integer-fix, we choose $\gamma = 10^{-2}$ and the maximum number β of iterations in both methods is set to 100. Since being long run processes, they are not called regularly for every node. Instead, they will be called at branch-and-bound nodes whose depth levels are equal to 1 (mod 4).

Columns "DaF" and "NIT" in Table 3.3 provide us with the numbers of feasible solutions found by the heuristics: dive-and-fix and near-int-fix, respectively. The first almost cannot present any solution. However, the second is quite helpful. This is likely due to the fact that fixing variables within a range is better (more feasible) than fixing a pre-specified number of variables. In all cases, the heuristics found out a half of the total number of all feasible solutions which have been found throughout the runs.

Cutting plane generation

In each iteration of the cutting plane phase, many violated inequalities are generated, but all should not be included into the current model. We choose a logical approach to select ones which violate the linear relaxation point x^* mostly. To do that, all algorithms for generating clique and odd cycle inequalities are performed to create a set of potential inequalities. Algorithms which can separate many violated inequalities, such as the row-lifting, greedy and GLS algorithms are controlled to produce a limited number of inequalities. We set this number to 5 for each separation iteration. Certainly, for the recursive smallest first algorithm which consumes an exponential computation time for optimality, we should limit the execution by setting $\alpha = 16$ and $\beta = 4$. Although the arrangement of generating many cutting planes and choosing only few of them spend a lot of the CPU time, this way helps generate stronger inequalities. Furthermore, the pool separation is applied before calling the cutting plane separation. Note that all used cutting planes are global.

Columns "clique" and "odd cycle" in Table 3.3 give us experimental results obtained from the execution of the branch-and-cut code using the default strategy. Row-lifting and greedy algorithms generate many clique inequalities as expected. The numbers

of the clique inequalities generated by the row-lifting, greedy, and RSF algorithms are presented in the columns "RF", "GR" and "RSF" respectively. Only RSF gives a small number of cutting planes. However, it is quite easy to tune RSF to an exact search by controlling its two parameters α and β. However, one should remember that large values for α or β will be computationally expensive. One thing should be commented on the column "m^+" of the table. The implementation mentioned in the previous paragraph makes it easy to understand why few cutting planes are really added to the integer models. We only choose most violated inequalities. The separation time takes only a small part of the total computation time (presented in the column "$\%_{\text{Sep}}$" by percentage). The code only spends much time in the separation step for problem "kl02". This is explained by the large number of variables in comparison with the number of constraints. This will bear strongly on algorithms which try to lift zero-valued variables into core inequalities (namely, row-lifting and greedy). An opposite outcome occurs with problems "aa01" and "aa04" which have many constraints but not such a large number of variables.

The same phenomenon also occurs with Vietnam Airlines test problems and randomly generated problems. Most of computation time is used for the linear relaxation and preprocessing phases.

Other aspects

An LP-based branch-and-bound code spends much time in computing linear relaxations. This also happens in our case in which column "$\%_{\text{Lp}}$" of Table 3.3 shows a large percentage of computation time involved in CLP solver, especially for the problem "aa01". Since the thesis does not go into details of linear programming, only default settings of the linear solver are used. Certainly, ABACUS provides a support of fast re-optimization (also called "warm start") and sets the dual simplex method to the linear solver by default. (Note that the dual simplex is suitable for a branch-and-cut code.) However, the computation time still depends strongly on the linear solver itself. It is expected that a commercial LP solver, such as CPLEX, XPRESS will deliver a much better performance. Another approach to enhance the linear relaxation time is to implement a special solver for the linear programming problem of the set partitioning model. This approach is mentioned by Hu and Johnson (1999) and implemented in a set partitioning solver of Linderoth et al. (2001). The *Volume algorithm* suggested by Barahona and Anbil (2000) is also a choice as a good solver for this kind of difficulty.

Note that, the linear programming time shown by ABACUS is not only due to the linear relaxation, but also the time required to set up linear programming problems. The sparse constraint matrix is not stored directly in a data structure which is preferred by the LP solver. It is built from the set of active variables and constraints as governed by ABACUS. Moreover as mentioned in the data structure section, we must store the matrix in a sparse format to save memory. ABACUS will restore its format before transferring the data to a linear solver. Due to the design which supports several

solvers, ABACUS spends much time to do such a task. A special design for a specific solver could improve much computation time (Borndörfer, 1998, Part II). The time for the linear initialization is quite high with large size test problems, such as "nw04", "us01", and Vietnam Airlines test problems. In such cases, it takes about half of "$\%_{\mathrm{Lp}}$". (Note that the linear relaxation time and linear initialization time are not shown separately in the table.)

Another overhead is due to other auxiliary parts of the branch-and-cut code. It can cost from 10 to 20 percent of the total computation time. With test problems having a large number of variables, this miscellaneous time increases noticeably. Below is the list of important parameters which are set to ABACUS.

branching	padberg-rinaldi ($\alpha = 0.25$)
`NBranchingVariableCandidates`	10
pool separation	yes
`MaxConAdd`	1
`MaxIterations`	20
`TailOffNLps`	5
`FixSetByRedCost`	yes
`VariableEliminationMode`	`ReducedCost`
`ConstraintEliminationMode`	yes
`EliminateFixedSet`	yes
`EliminateFixedSet`	yes

One unsolved big problem relates to memory capacity. If we increase the number of flight legs to more than 300, the branch-and-cut code will overload the available computer memory. If we want to solve a bigger crew pairing problem, we should consider a new effective storage method (e.g., to compress the data). One solution for this problem is to use a heuristic approach mentioned in Section 2.3.1. If we understand more about the characteristics of a specific crew pairing problem, it is easier to create a heuristic to replace current pairings by new ones to avoid local optima. Another way is to use a different approach which will be discussed in the next chapters. However, the branch-and-cut approach is quite helpful to solve small and medium crew pairing problems to optimality. Using a general framework, the set partitioning solver with the selected good algorithmic features can give a quite good performance.

Name	$\overline{z}$	B&B	Lp	clique			odd	m^+	n^-	m^-	heuristics		time					
				RL	GR	RSF	cycle				DaF	NIF	$\%_{\text{Pre}}$	$\%_{\text{Sep}}$	$\%_{\text{Heu}}$	$\%_{\text{Bra}}$	$\%_{\text{Lp}}$	t_{total}
aa01	56137	143	686	673	561	5	640	549	603	413	0	3	8.33	1.13	18.67	1.60	70.34	0:17:13
aa04	26374	303	1180	801	655	24	839	955	844	408	0	3	10.49	1.91	13.95	2.07	71.63	0:16:59
kl02	219	21	73	223	84	19	248	62	0	0	1	1	41.85	1.22	5.44	1.54	46.53	0:03:41
nw04	16862	105	446	969	577	162	76	341	0	3	1	1	25.26	26.86	5.68	0.55	31.97	0:12:29
us01	10036	9	65	300	167	49	157	60	0	0	0	0	13.40	4.46	5.87	0.90	59.97	1:10:12
*vn320r	66700	1	1	0	0	0	0	0	0	0	0	0	17.15	0.00	0.00	0.00	67.17	0:14:28
vn321	38870	1	1	0	0	0	0	0	0	0	0	0	77.38	0.00	0.00	0.00	13.41	0:01:09
vnAT7	22260	1	1	0	0	0	0	0	0	0	0	0	11.04	0.00	0.00	0.00	67.56	0:18:59
vncpp1	500943	17	44	132	68	9	179	35	0	0	0	0	35.29	1.21	2.38	1.23	53.69	0:10:00
vncpp2	697445	3	20	61	57	0	84	17	5	39	0	1	23.77	0.86	23.57	0.29	42.70	0:06:26
vncpp3	494154	13	39	149	57	0	165	30	392	60	0	1	35.77	1.11	3.48	0.94	51.83	0:08:03
vncpp4	506006	101	323	276	208	15	257	224	7328	983	0	3	12.60	0.77	14.49	2.08	67.49	0:39:06
aa01	56137	157	834	659	624	26	585	677	267	145	0	1	6.75	1.58	11.03	1.83	78.37	0:12:48
aa04	26374	691	3025	1998	1159	22	2332	2336	742	415	0	3	8.62	3.41	10.76	2.14	74.01	0:25:16
kl02	219	11	38	131	55	7	159	29	466	10	0	0	46.11	1.24	2.85	1.14	40.66	0:01:52
nw04	16862	223	1163	2870	1685	479	94	940	95	5	1	1	16.18	45.03	4.71	0.30	23.80	0:23:42
us01	10036	13	90	357	183	58	223	77	362	5	0	1	13.14	5.61	6.04	0.74	58.47	1:12:26
vncpp1	500943	31	69	201	116	15	260	52	280	170	0	2	20.71	1.36	20.42	1.32	51.27	0:13:49
vncpp2	697445	3	20	61	57	0	84	17	5	39	0	1	19.57	0.71	33.54	0.33	38.57	0:07:47
vncpp3	494154	31	77	187	86	9	193	50	1238	381	0	3	16.12	0.87	16.02	1.89	61.23	0:13:55
vncpp4	506006	165	451	276	221	20	248	286	3005	971	0	0	3.98	0.67	9.06	2.42	81.33	0:43:42
aa01	56137	157	834	659	624	26	585	677	267	145	0	1	6.81	1.56	10.98	1.84	78.36	0:12:43
aa04	26374	691	3025	1998	1159	22	2332	2336	742	415	0	3	8.67	3.40	10.67	2.14	74.07	0:25:09
kl02	219	11	38	131	55	7	159	29	466	10	0	0	46.11	1.24	2.83	1.18	40.79	0:01:52
nw04	16862	223	1163	2870	1685	479	94	940	95	5	1	1	16.27	44.99	4.70	0.30	23.77	0:23:35
us01	10036	13	90	357	183	58	223	77	362	5	0	1	13.07	5.65	6.11	0.74	58.44	1:12:34
vncpp1	500943	17	43	181	117	12	190	34	70	16	0	2	27.65	1.16	19.50	1.02	45.36	0:11:34
vncpp2	697445	3	20	61	57	0	84	17	5	39	0	1	19.68	0.71	33.32	0.32	38.66	0:07:44
vncpp3	494154	19	50	194	98	3	178	40	0	0	0	1	24.37	1.06	8.78	1.89	60.20	0:11:42
vncpp4	506006	101	324	276	208	15	257	225	7328	983	0	3	12.59	0.76	14.47	2.09	67.81	0:39:08

Table 3.3: Computational results of the branch-and-cut code with default strategy

Chapter 4

A Branch-and-Price Approach

In the previous chapter, we have been faced with the difficulty of having a very large number of variables. Although the integer programs (set partitioning problems in this case) can be solved with a few number of branch-and-cut nodes, it is not efficient to keep all variables in memory. In this chapter, we will consider a different approach which is based on a method that generates dynamically variables as needed. The method is called *column generation*. As implied in its name, the method deals with linear programming problems in which a large number of columns are listed implicitly and are generated on demand.

Column generation was originally suggested by Ford and Fulkerson (1958) for the multi-commodity flow problem. In 1960, Dantzig and Wolfe applied it to problems which are decomposed from more difficult integer linear programming problems. If problems can be partitioned into "hard" and "easy" sets of constraints, Dantzig-Wolfe decomposition is a suitable solution. Gilmore and Gomory (1961) used a similar idea to solve the cutting stock problem successfully. In recent decades, column generation is widely used in combinatorial optimization, especially in routing and scheduling problems which often have a huge number of variables in practical applications. Desrosiers et al. (1984) combined branch-and-bound and column generation to solve the vehicle routing problem with time windows. The combination is called *branch-and-price* by Savelsberg. Barnhart et al. (1998) give us a good introduction on branch-and-price and its techniques to deal with integer programs. Another survey is presented by Desrosiers et al. (1995) in which they focus on methods for solving the subproblems which are decomposed from the main problem.

4.1 Column Generation Method

Many combinatorial optimization problems can be easily transformed to integer linear programs which consist of a large number of variables. The set partitioning and set covering models can be the result of those transformations mentioned in Chapter 3. Dantzig-Wolfe decomposition also results in problems containing a large number of variables which is proven to give a stronger bound than the original model. This is

quite important to use within a branch-and-bound framework because the increase of dual bounds (in minimization problems) is accelerated. Another possible reason for using such models is that they can break the symmetry which often happens in compact formulations, and is an origin of the poor performance of numerical algorithms. Working directly with a large set of variables is not, in many cases, a clever approach. Instead, the column generation phase is only performed when needed. This reduces the memory usage in order to solve very large problems. A column generation approach decomposes the problem into a *master problem* and a *pricing subproblem*. In many cases, the pricing subproblem turns out to be easily solved due to its special structure. For example, the subproblem of the cutting stock problem is the famous *knapsack problem*. Any additional set of complex constraints from real world applications can be embedded easily. This will be seen in the process of solving the crew pairing problem discussed later. Moreover, it is more convenient to attack a separate pricing subproblem by different methods or techniques.

4.1.1 Column generation for linear programs

Consider the linear programming problem

$$\text{(MP)} \qquad \begin{aligned} z^*_{\text{MP}} = \quad & \min \quad c^\top x \\ & \text{s.t.} \quad Ax = b \\ & \qquad\; x \geq 0, \end{aligned} \tag{4.1}$$

which shall be called the *master problem*. Similar to notations in Chapter 1, let M and N be the index set of constraints and variables, respectively. In our context, we assume that N is so large that it needs to be handled by column generation. At iterate k, column generation works only on a subset of variables (also called "active" set of variables). We denote its index set by $N^k \subseteq N$. The linear programming problem associated with this subset is called the *restricted master problem* as following

$$\text{(RMP)}^k \qquad \begin{aligned} z^{*k}_{\text{RMP}} = \quad & \min \quad c_{N^k}^\top x_{N^k} \\ & \text{s.t.} \quad A_{N^k} x_{N^k} = b \\ & \qquad\; x_{N^k} \geq 0. \end{aligned} \tag{4.2}$$

It can be thought as derived by setting the other variables of (MP) to zero. In the revised simplex method, at each iteration, the method tries to find a variable to enter the current basis. In a similar behavior, a column generation method prices out a variable with a negative reduced cost. The only difference of column generation is that the pricing step is now modelled as an optimization problem:

$$\text{(PR)}^k \qquad r^k = \begin{array}{ll} \arg\min & c_j - {u^{*k}}^\top A_{\cdot j}, \\ j \in N & \end{array} \tag{4.3}$$

where u^{*k} is the solution of the associated dual program of $(\text{RMP})^k$. Rather than examining each index j separately, we can treat all indices in N implicitly if (PR) can

be modelled as a special problem. It is quite obvious that the optimal primal basis of the restricted problem is also primal feasible to the master problem. Therefore, after adding the new variable r into (RMP), we can continue to solve it quickly by a primal simplex method. Remember that the "warm start" technique depends strongly on a real implementation of the simplex method. A standard column generation method is outlined in Algorithm 4.1.1.

Algorithm 4.1.1 A standard column generation method

1: Find an initial index column set N^0 for the restricted master problem which must be feasible.
   $k \leftarrow 0$.
2: Solve the $(\text{RMP})^k$.
3: Solve the pricing subproblem $(\text{PR})^k$.
4: **if** $\bar{c}_{r^k} = c_{r^k} - u^{*k\top} A_{.r^k} < 0$ **then**
5: $\quad N^{k+1} \leftarrow N^k \cup \{r^k\}$.
6: $\quad k \leftarrow k+1$. Then go to 2.
7: **end if**

The finite termination of the algorithm is easily seen due to the finite set N of variables and the finite termination in solving $(\text{RMP})^k$. The column generation method will stop if it guarantees that all columns have been priced out correctly. This means that there is no variable having negative reduced cost (in a minimization problem). When this criterion is satisfied, the final dual solution u^* is also the dual solution of the master problem. Then, (x^*, u^*) is also the optimal solution of the master problem.

4.1.2 Dantzig-Wolfe decomposition principle

Matrices of large scale problems often show some special structures. For example, their sub-matrices possibly correspond to particular problems which are easier to solve. They are the target of decomposition methods which often try to decouple constraints or fix variables in order to obtain independent problems that can be dealt by certain specialized algorithms. Dantzig-Wolfe (1960) suggested a method to decompose the original problem in primal form.

Consider a (IP) as defined in Chapter 1 in which fractional variables are dropped out of the model. Suppose that the set of constraints of the problem can be split into 2 groups: one of "easy" constraints and one of "hard" constraints as follows:

$$\text{(IP)} \qquad \begin{array}{rll} \overline{z}_{\text{IP}} = & \min & c^\top x \\ & \text{s.t.} & A^1 x = b^1 \\ & & A^2 x = b^2 \\ & & x \in \mathbb{Z}^n_+. \end{array} \tag{4.4}$$

Note that the easiness of these constraints means that a problem with only these constraints can be solved easily due to its special structure. Assume $A^1 = b^1$ are "easy"

constraints, and $A^2x = b^2$ are "hard" constraints. Let $W = \mathbb{Z}_+^n \cap \{x : A^2x = b^2\}$. Then we can rewrite (4.4) as

$$\begin{array}{ll} \min & c^\top x \\ \text{s.t.} & A^1x = b^1 \\ & x \in W. \end{array} \tag{4.5}$$

If W is bounded, we can easily seen that W can be represented by a finite set of points, say $W = \{x^l : l \in L\}$. This is the basic idea of Dantzig-Wolfe decomposition. If W is unbounded, a result of Minkowski and Weyl (see Nemhauser and Wolsey (1988)) says that conv(W) is a polyhedron presented by a convex combination of a finite set of points and a linear combination of a finite set of rays. Therefore, a column generation form is still possible even if W is unbounded. More details can be found in Vanderbeck (1994), Goldfarb and Todd (1989). For simplicity, we only consider a bounded and nonempty W because this is the case in crew pairing problems. Then, any point $x \in W$ can be represented by a convex combination of $\{x^l\}$ as

$$x = \sum_{l \in L} \alpha_l x^l,$$

such that

$$\begin{array}{l} \sum_{l \in L} \alpha_l = 1 \text{ and} \\ \alpha_l \in \{0, 1\}, l \in L \end{array}$$

Dantzig-Wolfe decomposition applies this transformation to the program (4.5) to form the problem

$$\begin{array}{ll} \min & c^\top(\sum_{l \in L} \alpha_l x^l) \\ \text{s.t.} & A^1(\sum_{l \in L} \alpha_l x^l) = b^1 \\ & \sum_{l \in L} \alpha_l = 1 \\ & \alpha_l \in \{0, 1\}, l \in L \end{array} \tag{4.6}$$

Problem (4.6) is also called the column generation form of the master problem (IP). In the new problem, the "hard" constraints have been removed. However, this way of decomposition presents the problem with a huge number of variables since the polytope W can have a lot of points. After dropping the integrality condition of (4.6), we obtain its linear relaxation which is preferably solved by a column generation method. Let u be dual variables associated with the matrix A^1, v be the dual variable of the convexity constraint. After solving the associated restricted master problem of (4.6), the corresponding pricing subproblem of finding the least reduced cost in the column generation framework is

$$\begin{array}{ll} r = & \arg\min c^\top x^l - (u^{*\top} A^1 x^l + v^*). \\ & l \in L \end{array} \tag{4.7}$$

For a dual solution (u^*, v^*), a column of the form $\begin{pmatrix} Ax^r \\ 1 \end{pmatrix}$ which has a negative reduced cost will be added to the restricted master problem.

In a branch-and-bound framework, we often pay attention to the relaxation of integer programs. In our case, instead of working directly with W, we often study its convex hull $\text{conv}(W)$. Therefore, another relaxation of (4.4) is as follows:

$$\text{(DZLP)} \qquad \begin{array}{rll} z^*_{\text{DZLP}} = & \min & c^\top x \\ & \text{s.t.} & A^1 x = b^1 \\ & & x \in \text{conv}(W). \end{array} \tag{4.8}$$

Denote the optimal value of a linear problem obtained from (4.4) by dropping all integral conditions by z^*_{LP}. It is easily seen that

$$\overline{z}_{\text{IP}} \leq z^*_{\text{DZLP}} \leq z^*_{\text{LP}}.$$

This is due to the increasing feasible region size of the corresponding problems from left to right. This conclusion was presented by Geoffrion (1974) and also reclaims the reason why sometimes models with a large number of variables are preferred.

4.1.3 Branching

The idea of branch-and-price is complementary to that of branch-and-cut. Instead of generating rows to tighten a linear programming mode, branch-and-price produces columns to cut off the current optimal point of the restricted master problem. It easily happens that the feasible solution of the last relaxation is not feasible to the original integer programming problem. Therefore, when the optimum of the linear programming relaxation is reached, a further branching is possibly required for integer variables. The main difficulty lies in the branching phase. Other computational issues are discussed in more details in the work of Barnhart et al. (1998). In this thesis, they will be presented in the implementation.

Assume the column generation step has finished with a solution x^* to the relaxation of (4.4). A variable branching strategy will choose a variable, say x_i, to be branched to two branch-and-bound nodes:

$$x_i \leq \lfloor x^*_i \rfloor \text{ and } x_i \geq \lceil x^*_i \rceil.$$

The branching constraints must be moved into the pricing subproblem. For example, the pricing subproblem for the left node is:

$$\begin{array}{ll} \min & c^\top x^l - ({u^*}^\top A^1 x^l + v^*). \\ \text{s.t.} & l \in L \\ & x^l_i \leq \lfloor x^*_i \rfloor. \end{array} \tag{4.9}$$

Another approach is to branch a variable of the column generation formulation of the integer program. That means we force a fractional α_l to be an integer (0 or 1). However, with any kind of branching mentioned above, the structure of the pricing subproblem is broken. Remember that the pricing subproblem of most optimization

problems using column generation often is a network problem. Fixing a variable is not easily translated into the network.

Theorem 3.3.1 gives a solution to the rising problem above. Although the theorem was suggested in the context of set partitioning problems, it turns out to be effective in this area, especially for optimization problems which can be either formulated as or decomposed to a set partitioning problem. Remember the pair of branching constraints:

$$\sum_{j:A_{i'j}=1,A_{i''j}=1} x_j = 0 \text{ and } \sum_{j:A_{i'j}=1,A_{i''j}=1} x_j = 1.$$

They have a meaning that is quite interesting in the context of column generation. The main idea behind the branching is a co-relation between two rows i' and i''. When we use the set partitioning formulation to model a problem as mentioned in Chapter 3, rows correspond to elements of the ground set. In the network form of the pricing subproblem, the vertex set of its associated graph is also the ground set in the original model. In this case, a relation between two vertices is more natural and easier to be embedded into the graph. Now, we will consider in more details two generated branch-and-bound nodes. On the right node, there are only feasible columns which must have $A_{i'j} = A_{i''j} = 0$ or $A_{i'j} = A_{i''j} = 1$. In other words, two rows i' and i'' stay together in the set partitioning problem of the right node. Meanwhile, the left node only contains feasible columns which have $A_{i'j} = A_{i''j} = 0$ or $A_{i'j} = 0, A_{i''j} = 1$ or $A_{i'j} = 1, A_{i''j} = 0$. This means that the two rows are disjoint in the integer program of the left node. We see that the branching constraints have been embedded reasonably into child branch-and-bound nodes. This is an advantage of the Ryan-Foster branching method when being used with column generation.

4.2 Pricing Subproblem in Crew Pairing Problem

In order to solve the crew pairing problem, the branch-and-price approach discussed in this chapter employs the column generation idea to deal with a large number of feasible pairings which is a real difficulty to the branch-and-cut approach. To solve the relaxation of a branch-and-bound node, the branch-and-price approach decomposes the relaxed master problem into a restricted master problem containing a subset of all feasible variables

$$\begin{array}{ll} \min & c_{N^k}^\top x_{N^k} \\ \text{s.t.} & A_{N^k} x_{N^k} = e \\ & x_{N^k} \geq 0, \end{array} \tag{4.10}$$

and a pricing subproblem

$$\begin{array}{ll} \min & c_j - {u^{*k}}^\top A_{.j} \\ \text{s.t.} & j \in N, \end{array} \tag{4.11}$$

Note that the variables in the master problem correspond to pairings and the set partitioning constraints correspond to flights. The purpose of the pricing step is to

find negative reduced cost columns if they exist or to prove that there are no such columns. In the context of solving the crew pairing problem, it can be modelled as a shortest path problem with side constraints coming from the rules and regulations of airlines. Remember that this is true if we assume that the cost function is additive. It is the case when the cost of pairings is the sum of the non-negative cost of individual flights and the cost of resting times between flights.

In the case study of Vietnam Airlines, a suitable graph is created as follows. The vertex set V consists of all flights in the intended scheduling period. Note that, because the schedule is the same for following periods and the duration of a pairing is limited within a maximum number of days, additional vertices are needed for several following periods in order to cover all possible pairings. Two vertices $f_{i'}$ and $f_{i''}$ are connected if the arrival airport of $f_{i'}$ is the same as the departure airport of $f_{i''}$ and if the take-off time of $f_{i''}$ exceeds the landing time of $f_{i'}$ by at least a given so-called minimum ground time. We clearly obtain an acyclic directed graph. Denote the cost between the two flights by $d_{i'i''}$. It is easy to transform the cost of a flight into the cost of its outgoing arcs in order to consider only arc costs. The cost of an arc $(f_{i'}, f_{i''})$ now is $d_{i'i''} - u^{*k}_{i'}$ as depicted in Figure 4.1. Note that the flight $f_{i'}$ is associated with the dual variable $u^k{}_{i'}$ of the i'-th constraint. We create two additional vertices (called *super source* and *super target*) and then connect all vertices corresponding to flights departing from a given base to the super source and all vertices of flights arriving at that base to the super target. All these arcs are assigned costs as shown in the figure. Then, our pricing problem basically amounts to solving a shortest path problem on the acyclic graph with side constraints. The time involved in this step almost dominates the total computation time in many practical experiments. We will see this difficulty in computational results presented later.

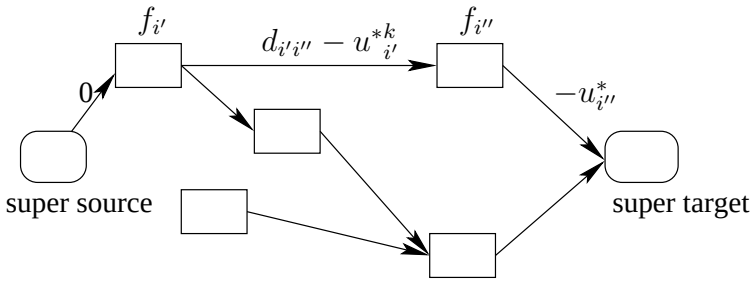


Figure 4.1: The network model of the pricing subproblem

A method for the pricing subproblem should work efficiently with the set of side constraints which can be extremely complicated and are changed over time. These side constraints are presented implicitly by N in (4.11). Widely-used methods which differ in their way of dealing with side constraints can fall into one of the four general categories below:

- Resource constrained shortest path problem,
- K shortest paths problem,

- Constraint programming,
- Hybrid approach.

In the next sub-sections, we will consider each method in order to find out those suitable for the considered crew pairing problems, especially the Vietnam Airlines case.

4.2.1 The resource constrained shortest path problem

This method is well-known in the context of column generation. It has been used successfully in solving various routing and scheduling problems. By its name, the resource constrained shortest path problem is an extension to the traditional shortest path problem with side constraints attached. From the complexity point of view, the shortest path problem can be solved in polynomial time, for example by using the well-known Dijkstra algorithm. But, the resource constrained shortest path problem is much harder due to the introduction of side constraints. With only one additive constraint, it turns out to be an $\mathcal{NP}$-complete problem (see Handler and Zang, 1980).

Definition 4.2.1. Given a directed graph $G = (V, A)$, a source node s and a target node t. A feasible path is constrained by a set of resource limits $\{U^r : r \in R\}$. Each edge (i, j) has a cost c_{ij} and a path passing it uses an amount of resource u_{ij}^r for the resource r. Cost and resource consumptions are assumed to be non-negative and additive along paths. The resource constrained shortest path problem is defined to find a least cost path from s to t obeying the resource constraints.

Time constrained routing and scheduling is one of the special applications of the resource constraint shortest path problem. In this area, an excellent description of algorithms and their applications can be found in Desrosiers et al. (1995). Different approaches towards the problem are also presented in Handler and Zang (1980).

Similar to algorithms for the shortest path problem, those for the resource constrained shortest path problem usually apply the idea of labelling (dynamic programming). Moreover, before expanding a path, all given resource constraints are checked to guarantee validity of the path. Therefore, the algorithm must keep track of resource usage while creating labels for vertices. Note that there are often many labels from s to each vertex because there is no guarantee that the best cost label among them will be feasible at the end. That is the reason why the algorithms sometimes are called *multi-labelling* algorithms.

It is obvious that the number of labels grow significantly from vertex to vertex. Therefore, a mechanism to remove redundant labels should be performed. Consider two labels l_1 and l_2 on a vertex v whose usage of resource r are l_1^r and l_2^r, respectively. Costs for each label are c_1 and c_2. If $c_1 \leq c_2$, and $l_1^r \leq l_2^r$, we say l_1 dominates over l_2 on resource r. This means that we can remove dominated labels without impact on the optimal solution. However, there is a set of resource constraints. Hence, we can only remove label l_2 if $c_1 \leq c_2$ and $l_1^r \leq l_2^r$, for all $r \in R$. It can be said that determining

Algorithm 4.2.1 A simple forward dynamic programming algorithm

1: $L_1 \leftarrow \{(0,0)\}$.
2: **for all** $i = 2 \rightarrow |V|$ **do**
3: **for all** $(i,j) \in A$ **do**
4: **for all** $(c, l^R) \in L_i$ **do**
5: **if** ($l^R + u_{ij}^R \leq U^R$) $\wedge$
($(c + c_{ij}, l^R + u_{ij}^R)$ not dominated by any label in L_j) **then**
6: Remove all labels in L_j dominated by $(c + c_{ij}, l^R + u_{ij}^R)$.
7: $L_j \leftarrow L_j \cup \{(c + c_{ij}, l^R + u_{ij}^R)\}$
8: **end if**
9: **end for**
10: **end for**
11: **end for**

whether a label is dominated by another label depends strongly on a specific set of side constraints.

A simple forward recursive algorithm for the resource constrained shortest path problem is introduced in Algorithm 4.2.1. Assume an acyclic directed graph $G = (V, A)$ modelling the pricing subproblem of a crew pairing problem. The vertices can be easily put into a partial order. In the algorithm, we denote by L_i the set of labels on vertex i. A label in L_i is a pair of the resource usage vector and the cost from s to i. There are several techniques to improve the performance of the resource constrained shortest path algorithm:

- *Pulling* technique (Desrochers and Soumis, 1988): Because most label algorithms use a *forward* recursive procedure such as the one shown in Algorithm 4.2.1, which updates the label list on the successors of the current vertex. The list will be modified several times because there are several incoming adjacent vertices. The pulling technique only updates the list of a vertex once by considering all its predecessors.

- *Pre-optimization* on resources: the process finds the minimal usage t_i^r of resource r from every vertex i to t. This can be done by solving several shortest path problems which consider the resource consumption as arc cost. By this way, a label on vertex i can be removed if $l^r + t_i^r > U^r$ for some r. The early pruning of labels can help reduce the memory usage and improve the performance significantly.

A problem that makes the resource constrained shortest path approach too difficult to apply for the crew pairing problem relates to the requirement of the additive property of resource consumption. Many resource kinds cannot be presented as or modified to an additive function. A limiting factor also comes from the fact that the algorithms for this problem only finds an optimal path. Therefore, if the best path turns out to be infeasible to non-additive constraints, we will have no information to find other

alternative paths quickly. The algorithms are only helpful if *all* side constraints are additive, which rarely happens.

4.2.2 K shortest paths problem

It is considered as a generalization of the shortest path problem, in which not one but several paths are generated.

Definition 4.2.2. Given a direct graph $G = (V, A)$, and a given number k. Each arc (i, j) has a cost c_{ij}. Then the k shortest paths problem is to find k shortest paths in the graph from a source to a target.

Since algorithms for the problem can produce many paths, we have more candidates for checking the feasibility of a path. If the shortest path is infeasible to some side constraints, the second, the third, or so forth can be used instead. Therefore the idea to employ the k shortest paths problem into solving the pricing subproblem is to temporarily forget side constraints and to find k shortest paths. These paths are ranked with respect to their costs. After that, all side constraints will be considered to check the feasibility of the paths. By doing so, we can easily deal with not only non-additive constraints, which cannot be overcome by the resource constrained shortest path approach, but also more complicated airlines constraints. One application of this approach can be found in the Carmen systems. Similar to the shortest path problem, the k shortest paths problem has also been well studied. One of the recent breakthroughs in computational complexity was presented by Eppstein (1998). In the paper, the author also gives a brief description of related works to solve the problem.

The k shortest paths algorithm presented by Jimenéz and Marzal (1999) is used to find k shortest paths for the side constraint checking. The idea behind the algorithm is to solve a set of recursive equations, which generalize the Bellman equations for the single shortest path problem. Denote by $L^k(v)$ the length of the k-th shortest path from s to v, and by $\pi^k(v)$ the path itself. The notation $L(\pi)$ means the length of the path π. The set of equations is presented as follows:

$$\begin{aligned} L^k(v) = &\begin{cases} 0 & \text{if } k = 1 \text{ and } v = s \\ \min_{\pi \in C^k(v)} L(\pi) & \text{otherwise} \end{cases} \\ \pi^k(v) = &\begin{cases} s & \text{if } k = 1 \text{ and } v = s \\ \arg\min_{\pi \in C^k(v)} L(\pi) & \text{otherwise.} \end{cases} \end{aligned} \tag{4.12}$$

$C^k(v)$ is the set of possible paths for choosing the k-th shortest path from s to v and is computed as follows:

$$C^k(v) = \begin{cases} \{\pi^1(u).v : u \in \Gamma^{-1}(v)\} & \text{if } k = 1, v \neq s \text{ or } k = 2, v = s \\ (\{C^{k-1}(v) \setminus \{\pi^{k'}(u).v\}) \cup \{\pi^{k'+1}(u).v\} & \text{otherwise } (\pi^{k-1}(v) = \pi^{k'}(u).v). \end{cases} \tag{4.13}$$

The so-called *recursive enumeration algorithm* merely reduces the candidate set $C^k(v)$ to obtain the set for the $(k+1)$-th shortest path. The algorithm, with the support of special data structures, has the complexity of $O(m+kn\log(m/n))$ after the shortest path from s to every vertex has been produced.

Although bypassing the obstacle of complex constraints, the k shortest path approach now faces a different problem. It seems that generally there is no method to determine which k to guarantee that there exists at least one feasible path in the k shortest paths that have been found. If one of the k paths is feasible, we make sure that a shortest feasible path has been found. If not, another exact method must be applied to determine the shortest feasible path.

4.2.3 Constraint logic programming

As mentioned in Sections 4.2.1, 4.2.2, the main difficulty in airline crew scheduling problems (crew pairing problem and crew rostering problem) is to deal with a set of many complicated rules and regulations. In the resource constrained shortest path approach, we must have additive constraints or must model rules as additive constraints. Furthermore, embedding rules into flight networks is a hard task in terms of programming and the design of an implementation must be thought about carefully beforehand in order to avoid the problem of adding new rules. As mentioned in Section 4.3.3, it is inefficient and hard to bring an ordinary variable branching strategy into a branch-and-price framework because the variable fixings will break the special structure of the pricing subproblem. In the k shortest paths approach, we must determine k in order to include at least one feasible path in a set of k shortest paths. In both cases, these troubles will lead to poor performance or lack of memory. *Constraint logic programming* (CLP) comes into play with an ability to present these rules quickly and easily.

Clearly, it is more comfortable to use the declarative programming model of constraint programming to represent airline rules, such as one supported in Prolog. Let $\mathcal{F}_b$ be a flight set that is to be scheduled in an intended schedule period. Because the schedule is the same for next periods and the duration of a pairing is limited within a maximum number of days, we repeat $\mathcal{F}^b = \{f^b_1, \ldots, f^b_{|\mathcal{F}^b|}\}$ several next periods so that all possible pairings will be covered, obtaining a new flight set $\mathcal{F} = \{f_1, \ldots, f_{|\mathcal{F}|}\}$. This set will be used as a base domain for flight variables. We use a tuple $P = (x_1, \ldots, x_{|P|})$ to present a pairing variable. In such a case, a flight variable $x_i \in \mathcal{F}$, except that $x_1 \in \mathcal{F}^b$ because the first flight must be within the schedule period. With these notations, we can easily present all airline rules. This is also one of the modelling approaches to apply CLP into solving the pricing subproblem.

For example, in order to represent the rule that the length of a feasible pairing cannot exceed `MaxLegs`, the resulting CLP rule in ECL^{i}PS^e(a CLP programming environment) can be as below:

```
rule_length( P ) if length( P ) < MaxLegs.
```

The rule to present the consecutiveness of airports in a pairing is:

```
rule_connect( [_] ).
rule_connect( [X1,X2|T] ) if
  X1.arrivalport = X2.departureport,
  rule_connect( [X2|T] ).
```

The objective function of the pricing subproblem is possibly presented as a CLP rule in which the reduced cost of a pairing must be less than 0. Certainly, since CLP is simply an enumeration method, we can obtain many negative reduced cost feasible columns rather than the most negative cost column. However, it is possible to control the process of fixing variables in a CLP engine. One problem of CLP is the poor performance if we do not add any additional constraints to accelerate the propagation of a variable fixing.

CLP includes propagation mechanisms to exclude impossible alternatives in solution spaces. The fact of fixing a variable will be propagated to other variables in order to narrow down the search space. This seems useful for our crew pairing problem in which variables have finite domains and strong relations. CLP has emerged in recent years withing the operations research community to provide a powerful tool to solve combinatorial optimization problems. A large number of users have used CLP to solve crew scheduling problems. Some of them are Guerinik and Caneghem (1995), Fahle et al. (1999), Yunes et al. (2000).

4.2.4 Hybrid approach

A hybrid approach, by its name, employs different methods to create a strong solution for a specific problem. In order to combine some of the previous methods, we need an exact method for a general crew pairing pricing subproblems as the last resort. The CLP approach is a solution to a general pricing subproblems. In addition, exhausted enumeration is always the choice in this case.

Enumeration has been considered as the worst solution to many combinatorial optimization problems. However, in our case, it has the following advantages:

- An implementation of an enumeration strategy is easily done in a short time. It is possible to put the rule checking in a separate block. Certainly, simple rules should be considered in the process of extending a path in enumeration. This will help delete a lot of candidates without (or with small) impact on the total performance.

- An implicit enumeration solution like branch-and-bound is possibly preferred to an explicit enumeration. Because the purpose of the pricing step is not only to find a negative reduced cost column, but also to prove that no such column exists. Therefore, with a bounding process in an implicit enumeration, we can avoid combinatorial explosion. Furthermore, if the implicit enumeration is used as the second choice after one of the previously mentioned methods fails, we can re-use information obtained from the first method. For example, after applying

unsuccessfully a k shortest paths algorithm, the bound given by the k-th shortest path is used as the lower bound in the branch-and-bound search.

- In a clearly structured algorithm like an implicit enumeration, heuristics can be embedded with little effort. Problem specific heuristics are quite useful in solving large optimization problems. However, in this case, the thesis focuses on considering a general framework which allows more rules to be inserted rather easily. Thus, only general heuristics will be used in the implementation.

After having an exact method, we can combine all methods to solve the pricing subproblem. The discussion of the hybrid method is postponed to the computational part.

4.3 Computational Issues

4.3.1 Test problems

Unfortunately, we cannot use the problem set of Hoffman and Padberg (1993) because all its problems have been transformed into set partitioning problems. Moreover, if we had their original flight sets, we would not be able to obtain the airlines rules for them. Instead, the airlines rules of Vietnam Airlines are used to test the branch-and-price implementation.

Four sets of crew pairing problems will be used in experiments. The two first sets have been described in Chapter 3. They are the set of timetables of Vietnam Airlines and the set of randomly generated timetables operating on an extended network of Vietnam Airlines. The third set is generated randomly from a flight network depicted in Figure 4.2. Aircrafts operate among 21 airports of which there are seven home bases for crews. Four timetables have been generated from the network following the generation scheme described in the previous chapter. The name of these timetables has a prefix "vircpp" and each has 284 flights. The characteristics of these crew pairing problems are also the same as those of the two first sets. The fourth set (having a prefix "vircppl") also operates on the same network but with more flights than that of the third. Since combining 458 flights into pairings creates very large set partitioning problems which cannot be loaded into the memory of the test computer, we follow the same idea mentioned in Section 3.3.6 to deal with the problem "'vn320". All problems in Table 4.1 marked with a "*" at the end of the name are processed in this way. In columns "n" and "nnz" of these problems, there is a pair of values: the first is of the associated reduced problem and the second is of the original problem.

The problems in the two additional sets are also solved by the branch-and-cut code with the default settings. The computational results are briefly shown in Table 4.1. It is transparent that $\overline{z}$ of the "vircppl" problems only gives us a non-optimal solution.

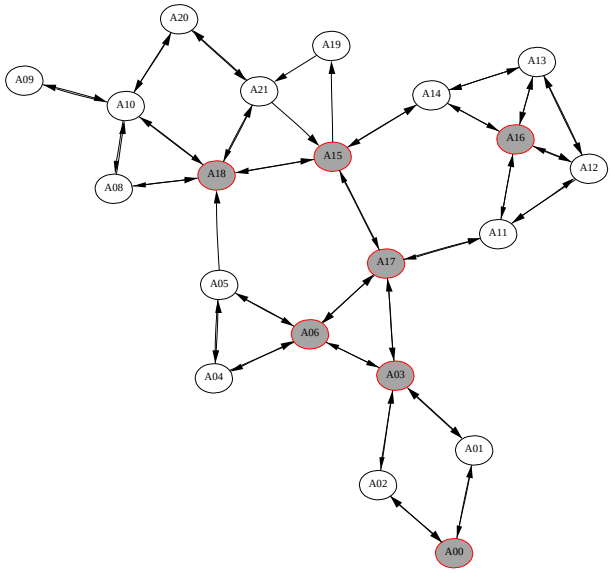


Figure 4.2: The flight network for the third set of crew pairing test problems

Name	m	n	nnz	t_{Enu}	branch-and-cut			
					$\overline{z}$	B%B	Lp	t_{total}
vircpp1	284	168542	945118	0:00:47	107429	11	78	0:09:43
vircpp2	284	125012	692533	0:00:36	134421	25	121	0:15:16
vircpp3	284	135601	751801	0:00:40	135116	5	11	0:04:43
vircpp4	284	137885	765612	0:00:39	137165	21	74	0:05:24
vircppl1*	458	338679/967731	1905757/5569753	0:19:35	142116	3	16	0:15:08
vircppl2*	458	265301/723364	1475911/4135015	0:10:30	176721	1	1	0:07:26
vircppl3*	458	286778/792719	1599326/4536834	0:11:29	180468	1	1	0:09:00
vircppl4*	458	294157/837538	1640275/4802571	0:14:23	170023	1	1	0:08:10

Table 4.1: Two additional sets of crew pairing problems, including their computational results of the branch-and-cut code

4.3.2 Initial solution

In a branch-and-price framework, the restricted master problem must be provided with an initial solution. How to create it strongly depends on the application. The initial solution for crew pairing problems in this thesis will be generated by a greedy heuristic which prefers to connect the nearest flight to an opening pairing. Note that we assume that the objective function of crew pairing problems is to minimize the total resting time between consecutive flights. One can also use the heuristics of Baker et al. (1979) to build up the initial set of pairings.

However, the process of generating pairings is highly constrained by a set of complicated airline rules. Therefore, we cannot easily obtain a feasible solution, which covers *all* flights. Furthermore, the pricing subproblem needs a good dual solution as an input. A common method in this case is to include high cost variables into the initial model. In this case, for each flight, an artificial pairing which contains only that flight is included. This idea is quite familiar in the context of simplex algorithms.

4.3.3 Branching

The implementation will follow the Ryan-Foster idea mentioned in Section 4.1.3. Since this branching requires a search for 2 flights i' and i'' on many possible pairs of them, we just perform the task on a limited number of pairs. The number is given in a parameter file under the name `MaxNoRFChecked` which is 100 by default. If the overhead of the branching phase is not too high, we can set this number to the maximum (i.e., to search on all pairs).

In any branch-and-bound based method, an optimal method often applies branching methods which create rather balanced trees. This also happens here with the Ryan-Foster branching. The fractional sum $\sum_{j:A_{i'j}=1,A_{i''j}=1} x_j^*$ of the relaxation gives us many possibilities. If we prefer to find the optimal solution, we can choose those pairs of flights having a sum near to 0.5. Otherwise, if we prefer to find a good feasible solution early, we can choose those having a sum near to 1. Remember to consider the new problem with the constraint

$$\sum_{j:A_{i'j}=1,A_{i''j}=1} x_j = 1$$

before the problem with the constraint

$$\sum_{j:A_{i'j}=1,A_{i''j}=1} x_j = 0$$

because the former is clearly stronger than the latter. In the implementation, the preferred value for the sum will be supplied by a parameter α.

4.3.4 Pricing subproblem

Solving the pricing subproblem is one of the main focuses in any branch-and-price based application solvers. In the previous Section 4.2, the thesis has presented a group of widely used methods for this problem. Since the thesis does not have enough space to present all computational results for all combinations of algorithms and running parameters, certain important observations will be shown to choose a suitable method for the problem.

Resource constrained shortest path problem

The pricing step must generate feasible pairings which start and end at a same home base. Hence, to be able to apply a resource constrained shortest path algorithm, we must decompose the problem according to home bases. An acyclic directed graph will be created for each home base in which flights starting from that base are connected to the super source, and flights ending at the base are connected to the super target (see Figure 4.1). Solving a resource constrained shortest path problem on the whole network covering all flights is not affordable in terms of computer memory for medium-ranged problems. Therefore, a different way of creating the graphs is used. Each flight

departing from a base will be used as the super source. This means that we only generate negative reduced cost pairings starting from that flight. Remember that we do not have to find the most negative reduced cost. Instead, the pricing step can be considered as either generating a number of pairings with negative reduced costs or proving no more such pairings.

Another point to be addressed is from which flight leg the pricing method will start in the next iteration of the column generation. If we always start from the same flight leg (e.g., the first flight of the schedule) in every iteration, we possibly only generate new pairings which are similar in structure to the pairings already contained in the current restricted master problem. Therefore, they will only slightly improve the integer programming model. In order to make the search more effective, the implementation changes the starting flight leg in every iteration. In reality, it starts from the flight leg right after the one where the previous iteration stopped.

As mentioned before, the resource constrained shortest path algorithms demand that all resources are additive. This is not satisfied by the test crew pairing problems which contain several kinds of non-additive resources. They are excluded from the set of airline rules for a correct comparison.

K shortest paths problem

The k shortest paths algorithm in use also depends on a labelling technique. So it has similar properties to the previous method. In addition, a feasible pairing cannot stop over at a home base. This means that we only generate fundamental pairings. Meanwhile, a long pairing which is combined by fundamental ones could have lower reduced costs than its elements, although it is infeasible. Thus, applying a k shortest path algorithm to the whole graph is inefficient. A solution for that is to remove arcs going out from flights which stop at the currently considered home base. Certainly, the whole problem is decomposed into small problems in which the super source is assigned each time to one flight starting from a home base.

Controlling the parameter k is quite crucial in the algorithm to find out pairings with negative reduced costs. In the implementation, after every iteration, k is increased in order to raise the possibility to produce a pairing for the restricted master problem. However, we do not have any conclusion when the k shortest paths algorithm has found no column. An exact pricing method must be involved to guarantee the optimality of the column generation method.

If we want to change the branch-and-price implementation to a heuristic, the pricing subproblem is not necessary to be solved to optimality. In this case, the k shortest path approach can be used without any further exact pricing method. Another good point is that we do not have to include airline rules into the graph.

Constraint logic programming

Vietnam airline rules and regulations are expressed quickly using the predicate structure of the constraint logic programming. Modelling a rule is not only implementing

an expression to check whether a pairing is feasible, but also improving the expression to utilize the propagation mechanism of CLP which remove impossible alternatives in solution spaces as a result of variable fixing. In order to improve the performance of the algorithm, a constraint is included in order to exclude prematurely pairings which cannot introduce negative reduced costs finally. The idea behind it is to reduce the domain of a flight variable of a pairing variable with the help of dual values and the minimum resting period between flights. In order to make search space quickly smaller, we use the first-fail search method. This means the flight variable with the smallest domain size will be instantiated firstly. It is still open to many search methods to be applied here.

Poor computational performance is a crucial problem of any constraint logic programming environment. This also happens with ECL^{i}PS^e. In the first attempt, the task of find negative reduced cost pairings is assigned to CLP. A series of thorough experiments have been made, but the performance is unacceptable. Therefore, the idea of applying CLP only for rule checking is chosen. This will be useful in real world crew pairing problems in which a large set of complicated rules must be modelled and modified dynamically with little effort. However, the performance of CLP is still extremely slow in comparison with `C++` rule checking (see Hoai et al., 2003). This is also the reason why the method is not included in this thesis.

Hybrid approach

Section 4.2 presents several possible reasons to use enumeration. The branch-and-price code considers an exhausted enumeration with additional heuristics. Like previous methods, the enumeration also iterates through flights in a round robin manner. For each starting flight, a depth first search is performed. Furthermore, there are the following additional features.

- If a pairing with negative reduced cost has been found, the method removes all flights contained in that pairings from the graph. This will reduce the graph size in future searches for further pairings. This is also reasonable because two pairings having some flights in common cannot belong to an integer feasible solution at the same time.

- The depth first search of a path can be stopped early if the path cannot produce a pairing due to either the non-negativeness or the infeasibility of its final paths. The idea is quite similar to the pre-optimization technique widely used in the resource constrained shortest path approach.

The enumeration can be used alone or in a hybrid approach. Generally, we have two types of pricing methods: inexact and exact. Obviously, the exact pricing method must be invoked finally in order to prove at least that there is no more pairing with a negative reduced cost. However, it is not necessary to call exact methods in the early iterations. A solution to combine these two types together is to employ a two-phase

approach. A fast inexact method is called first. If it generates certain pairings, the two-phase pricing method stops immediately. Otherwise, it brings up an exact method.

The code implements a hybrid framework which is simply to combine an inexact method with an exact one. In this case, the k shortest paths algorithm is used as the inexact method. The exact method will be selected between the resource constrained shortest path algorithm and a heuristic enumeration as mentioned above.

Experiments on the test problems have been performed with the different pricing methods. Since this section just want to find a good pricing method, the experiments are only executed without branching and their results are shown in Table 4.2. Notations used in the tables are the same as those used in Chapter 3. The most interesting column is "t_{total}". The heuristic enumeration delivers the best performance. A combination of the k shortest paths method and the heuristic enumeration also presents a good performance. Moreover, it needs less average numbers of column generation iterations ("Lp") and variables ("n^+"). Meanwhile, the resource constrained shortest path method gives a poor performance. Even working with a reduced set of airline rules, the method generates a very large number of labels. Vance et al. (1997) considered this problem and their solution is to change the branch-and-price implementation into a heuristic. It is clear that the number of iterations of the method is often small because it finds out the most negative reduced cost in every iteration. The table of the k shortest paths method only depicts the number of iterations that the method is still effective. The lower bound of this method cannot be used as that of the root node because not all pairings have been priced.

4.3.5 Primal heuristics

Reducing the gap between the dual bound and primal bound can be done by heuristics to find or improve feasible solutions. Obviously, choosing a suitable heuristic and controlling its running parameters depends strongly on the specific structure of an application. Three local search heuristics are included in the branch-and-price code to improve found feasible solutions.

- 2-opt heuristics: Figure 4.3 shows the idea of the 2-opt local heuristics. With the cost structure of Vietnam Airlines, we can obtain a better solution by replacing long resting times with shorter ones.

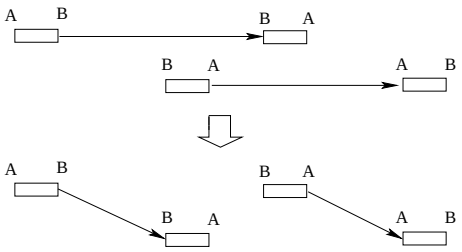


Figure 4.3: 2-opt heuristic

Resource constrained shortest path method

Name	$\underline{z}$	Lp	n^+	label	$\%_{\text{Pri}}$	$\%_{\text{Lp}}$	t_{Pri}/ite	t_{total}
vncpp1	500465	117	853	9795097	99.57	0.40	13.37	0:26:11
vncpp2	696546	115	893	5109541	99.19	0.75	7.94	0:15:21
vncpp3	493076	115	915	5082548	99.06	0.88	7.62	0:14:45
vncpp4	503021	150	1006	11292105	99.45	0.51	12.47	0:31:21
vircpp1	106409	109	900	10285642	99.65	0.32	14.75	0:26:53
vircpp2	132909	94	781	5257825	99.51	0.44	9.26	0:14:35
vircpp3	131745	103	863	5357666	99.56	0.39	9.64	0:16:37
vircpp4	136616	107	905	7950447	99.61	0.35	11.92	0:21:20

K shortest paths method

Name	$\underline{z}$	Lp	n^+	$\%_{\text{Pri}}$	$\%_{\text{Lp}}$	t_{Pri}/ite	t_{total}
vncpp1	645890	61	588	92.03	6.78	0.27	0:00:18
vncpp2	1573940	49	446	94.55	4.68	0.37	0:00:19
vncpp3	644313	47	445	92.60	6.34	0.32	0:00:16
vncpp4	1472880	48	470	93.16	6.01	0.29	0:00:15
vircpp1	338571	57	536	93.24	5.83	0.25	0:00:15
vircpp2	256568	51	478	93.10	6.03	0.29	0:00:16
vircpp3	155786	53	486	94.59	4.51	0.30	0:00:17
vircpp4	1319820	65	638	91.28	7.52	0.21	0:00:15

Heuristic enumeration

Name	$\underline{z}$	Lp	n^+	$\%_{\text{Pri}}$	$\%_{\text{Lp}}$	t_{Pri}/ite	t_{total}
vncpp1	500465	194	1864	89.34	10.43	0.77	0:02:47
vncpp2	696546	194	1883	89.77	10.01	0.91	0:03:17
vncpp3	493076	196	1899	88.43	11.33	0.73	0:02:41
vncpp4	503021	197	1923	84.98	14.72	0.59	0:02:16
vircpp1	106409	199	1755	97.07	2.83	1.81	0:06:12
vircpp2	132909	159	1541	95.36	4.41	1.00	0:02:46
vircpp3	131745	145	1322	96.24	3.55	1.03	0:02:35
vircpp4	136616	208	1873	96.50	3.39	1.55	0:05:35
vircppl1	138962	328	3196	97.55	2.39	6.05	0:33:54
vircppl2	172947	333	2815	99.07	0.91	13.14	1:13:37
vircppl3	179818	298	2880	97.66	2.27	5.75	0:29:16
vircppl4	169720	356	3228	98.50	1.47	10.83	1:05:15
sum		2807	26179			44.16	3:50:11

K shortest paths method→ Heuristic enumeration

Name	$\underline{z}$	Lp	n^+	$\%_{\text{Pri}}$	$\%_{\text{Lp}}$	t_{Pri}/ite	t_{total}
vncpp1	500465	147	1099	97.81	2.05	1.94	0:04:52
vncpp2	696546	144	1190	97.42	2.45	1.85	0:04:33
vncpp3	493076	167	1300	98.08	1.85	2.86	0:08:07
vncpp4	503021	167	1373	96.62	3.23	1.68	0:04:51
vircpp1	106409	142	1035	98.66	1.24	2.20	0:05:16
vircpp2	132909	125	988	97.49	2.32	1.26	0:02:42
vircpp3	131745	136	905	98.80	1.12	2.06	0:04:43
vircpp4	136616	202	1424	98.17	1.73	1.96	0:06:44

Table 4.2: Performance of different pricing methods to solve the root node of the crew pairing problems

- Scrolling heuristic: since the timetable is duplicated for the next periods, a flight can shifted and used in either the periods. Figure 4.4 demonstrates the idea of the heuristic.

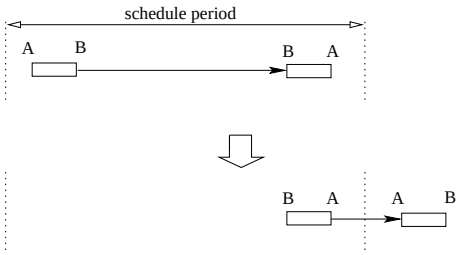


Figure 4.4: Scrolling heuristic

- Merging heuristic: most airlines design an operational network consisting of several main airports and many subsidiary ones. An aircraft flying to a subsidiary airport often returns to the starting main airport right after. This allows to merge short pairings together as shown in Figure 4.5. Remember that A $\neq$ C because a valid pairing must end at its starting airport immediately.

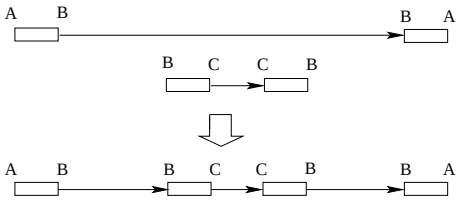


Figure 4.5: Merging heuristic

The three local improvement techniques above possibly give better feasible solutions, especially in the case of the Vietnam Airlines problems. It is also possible to use them in a meta-heuristics, such as simulated annealing. However, we should keep in mind that it is hard to define the neighborhood structure because there are many complicated airline rules. A solution is to relax some difficult rules similar to the ejection chains technique used by Rego (1998).

4.3.6 Computational results

The machine, the operating system and the compiler used in computation are the same as those in Chapter 3. ABACUS is still the framework for the branch-and-price code. Unlike the branch-and-cut approach, the branch-and-price approach does not have much difficulties with storage. The latter only deals with a small set of variables which currently belong to the restricted master problem. The variables, which correspond to pairings, keep the set of flight identifications and the constraints are the flights

themselves. Remember that the variables are locally valid, and generated dynamically by a pricing method.

Important running parameters have been set as shown in the table below.

Initial solution	greedy heuristic
Pricing method	heuristic enumeration
Pool separation	yes
`MaxVarAdd`	10
`VariableEliminationMode`	no
Reduced cost fixing	yes
Primal heuristics	yes
Branching	Ryan-Foster ($\alpha = 0.5$)
`MaxNoRFChecked`	maximum

The greedy heuristic is chosen to create the initial solution for the restricted master problem. This choice comes from the cost structure of Vietnam Airlines. As seen in Section 4.3.4, the heuristic enumeration is the fastest method which is selected as the pricing method in the next branch-and-price computations. The number of variables (pairings) generated each time is certainly not necessarily equal to 1. It should be large enough to utilize the time-consuming pricing step. However, a very large value of `MaxVarAdd` is not a good choice either. Doing so can make the whole algorithm gets stalled inside the pricing step. The pool separation could helpfully share the hard task imposed on the pricing method. Since the objective is to find the optimal solution, the parameter α of the Ryan-Foster branching is set to 0.5.

The features of fixing and eliminating variables/constraints of ABACUS are switched off because they can break the data structure storing the crew pairing problems. However, the reduced cost fixing is still re-implemented in the application code. Fixing a variable to 0 is not implemented because it is hard to translate the fixing into the graph of a pricing method. By contrast, fixing a variable to 1 can be translated efficiently into the graph. The fixing helps remove flights of the variable out of the graph, leading to a shorter execution time in the pricing step. In Table 4.3, we can see an extremely small percentage "$\%_{\text{Pre}}$" of computation time involved in this feature. The same situation also occurs with the local improvement heuristics though they contribute several enhancements of the primal bound ("heuristics").

Column generation

In all computations, about 90% of the total time is consumed by the pricing subproblem (shown in the column "$\%_{\text{Pri}}$"). The most important advantage of branch-and-price is to work only a small set of variables. Column "n^+" shows small numbers of variables added to the restricted master problem in all cases. Remember, that the initial solution only contains a few variables. Therefore, we do not worry about the memory problem in the branch-and-price approach. Furthermore, the small size of (RMP) also explains why the column "$\%_{\text{Lp}}$" is quite small in comparison with the branch-and-cut approach.

Name	$\overline{z}$	B&B	Lp	n^+	n^-	heuristics			time						
						2-opt	scroll	merge	$\%_{\text{Pre}}$	$\%_{\text{Pri}}$	$\%_{\text{Heu}}$	$\%_{\text{Bra}}$	$\%_{\text{Lp}}$	t_{Pri}/ite	t_{total}
vn320	64465	1	103	992	0	5	8	20	0.00	99.84	0.01	0.00	0.13	22.88	0:39:20
vn321	38870	1	26	100	0	0	1	2	0.00	99.12	0.00	0.00	0.64	1.11	0:00:29
vnAT7	22260	1	102	937	0	3	2	2	0.00	99.66	0.01	0.00	0.32	10.11	0:17:15
vncpp1	500943	9	225	1916	0	8	6	5	0.00	89.42	0.02	0.03	7.50	1.28	0:05:23
vncpp2	697445	11	235	1963	0	4	1	5	0.00	91.31	0.01	0.04	5.93	1.85	0:07:55
vncpp3	494154	193	978	4098	16	7	4	4	0.00	88.44	0.00	0.07	5.03	3.61	1:06:30
vncpp4	506006	95	595	2894	2	5	4	5	0.00	87.31	0.00	0.07	5.77	2.92	0:33:11
vircpp1	107429	39	361	2114	0	0	5	8	0.00	92.87	0.00	0.03	2.89	3.25	0:21:03
vircpp2	134421	85	334	1843	0	3	7	2	0.00	86.03	0.00	0.09	2.94	2.96	0:19:10
vircpp3	135116	21	306	1788	0	2	5	6	0.00	94.11	0.01	0.02	2.46	2.33	0:12:39
vircpp4	137165	35	420	2507	0	4	4	6	0.00	93.33	0.00	0.03	3.22	3.02	0:22:38
vircppl1	139344	97	587	3913	0	5	8	8	0.00	94.50	0.00	0.02	1.37	22.82	3:56:17
vircppl2	173538	293	1013	4522	0	5	6	11	0.00	93.00	0.00	0.02	1.07	27.70	8:22:56
vircppl3	179818	1	298	2880	0	7	15	15	0.00	97.61	0.02	0.00	2.32	5.68	0:28:53
vircppl4	169852	7	394	3436	0	0	4	4	0.00	97.88	0.00	0.00	1.43	12.91	1:26:36

Table 4.3: Computational results of the branch-and-price code

Branching

In spite of searching all pairs of flights for two flights satisfying the Ryan-Foster condition, the branching step needs little time (shown in the column "$\%_{\text{Bra}}$"). That is the reason why the parameter `MaxNoRFChecked` which could be helpful for huge timetables will be not used.

Unfortunately, we cannot apply the strong branching idea for the branch-and-price approach. There is only a small part of all variables in the restricted master problem. That is the reason why executing a number of simplex iterations cannot tell the possible improvement of the whole model. Most cases of the branch-and-price executation require more numbers of branch-and-bound nodes in comparison with the branch-and-cut code.

Although one only needs one branch-and-bound node for the problem "vn320", the branch-and-price code now can solve it to optimality using the same computer and compiler options. This is also quite useful in the crew pairing phase. Hoffman and Padberg (1993) present a set of crew pairing problems in which most of them require only one branch-and-bound node. The optimal solution of all problems obtained from Vietnam Airlines has been reached without branching. Do not be surprised why many enumerations have been invoked in the pricing step, but with a small time in comparison with the unique enumeration of the branch-and-cut code in the beginning. This is because the pricing subproblem has a dual vector to make the enumerations faster.

Now, we will see how the branch-and-price code works on the randomly generated test problems. It can be seen in Tables 4.3 and 3.3 that, with the size of all timetables in the sets "vncpp" and "vircpp", the branch-and-price approach is outperformed by the branch-and-cut approach. However, the method presented in this chapter has solved the problems in the set "vircppl" to optimality without overloading the computer memory. "vircppl3" and "vircppl4" requires a few branch-and-bound nodes to reach the optimal solution while "vircppl2" is hard to be solved. Besides the effective usage of memory, the branch-and-price code also has a stronger ability to determine more feasible solutions in its optimization process as observed during its running progress. This can be explained by the fact that working on a small number of variables, the code is easier to find a feasible solution to cover all flights.

Let's have a look on the column "Lp" of Tables 4.3 and 4.2. We see that about half the number of LPs are involved in solving the root node. This will be the topic of the next section which discusses how to reduce that number.

4.4 Stabilized Column Generation Method

Column generation has been used for solving a variety of scheduling applications because it is able to deal with problems which are modelled by a huge number of variables. In this scheme, the solution of the pricing subproblem is an important and difficult issue. In the experiments reported in the previous section, usually the time for pricing amounts to about 90% of the total computing time. The oscillation of dual points is the

main reason for instability in the standard implementation of this method, especially in the root node. One device to deal with the problem is to use the *trust-region* idea to stabilize the path of dual points.

The main approaches which stabilize the development of dual variables in order to accelerate the solution process are listed in the following. The Bundle methods (see Hiriart-Urruty and Lemaréchal, 1993) create a trust region with penalty to prevent the next dual point from moving too far. The analytical center cutting plane method (Goffin and Vial, 1990) takes an analytic center of a region in the dual function instead of having the optimal dual solution as next iterate. Another idea is the so-called primal-dual subproblem simplex method of Hu and Johnson (1999). The new dual point used for constructing subproblems is a convex combination of the previous dual point and the dual solution of the current subproblem. It also must be a dual feasible solution to master problem. Barnes et al. (2002) suggest an algorithm to deal with the degeneracy by finding a strictly improving direction for the current dual problem at the current dual point by solving a least-squares problem. An old method is the BoxStep method presented by (Marsten et al., 1975) which restricts the current dual point in a bounded region. This method has been generalized by a model of du Merle et al. (1999) and further investigated by Neame (1999), Ben Amor and Desrosiers (2003). The thesis will apply this idea which was only considered briefly in the study of du Merle et al. into solving the considered crew pairing problems. Borndörfer et al. (2001) discussed a similar approach of using the BoxStep model in duty scheduling.

In order to see the need of a stabilizing device, we revisit the standard column generation method. The dual of the column generation method is sometimes considered as Kelley's cutting plane method (see Kelley, 1960, Neame, 1999). Hence we equivalently speak of cutting planes in the dual space and of columns in the primal space. Therefore, in dual space, the pricing step is similar to the process of finding cutting planes that are violated by the current optimal point. The oscillation of dual points is a characteristic of the standard column generation methods. The instability which occurs if we do not apply the trust region idea can be seen easily. Because of not being bounded, u^{*k+1} can be far away from the current point u^{*k}, and also from the optimal dual solution. We call this behavior *unstable*. This leads to the possible poor performance of the standard column generation method with respect to creating redundant cutting planes (in the dual problem). More details about it can be found in Hiriart-Urruty and Lemaréchal (1993), Lemaréchal (2001).

In order to overcome this disadvantageous behavior, the possible values of the dual variables are restricted by defining some box (a trust region) around the current iterate (as in the BoxStep method) and ensure that the dual variables do not move too far away. The thesis will discuss the effect of this idea on the number of iterations as well as on the overall CPU time.

4.4.1 Generalized stabilized column generation method

In order to describe the approach for controlling the dual variables, we will consider a primal problem and its dual problem, respectively, in a form suggested by du Merle et al. (1999):

$$\begin{array}{lrcl} \min & c^\top x - \delta_-^\top y_- + \delta_+^\top y_+ & & \\ \text{s.t.} & Ax - y_- + y_+ & = & e \\ & y_- & \leq & \varepsilon_- \\ & y_+ & \leq & \varepsilon_+ \\ & x, y_-, y_+ & \geq & 0, \end{array} \tag{4.14}$$

$$\begin{array}{lrcl} \max & u^\top e - \varepsilon_-^\top v_- - \varepsilon_+^\top v_+ & & \\ \text{s.t.} & A^\top u & \leq & c \\ & -u - v_- & \leq & -\delta_- \\ & u - v_+ & \leq & \delta_+ \\ & v_-, v_+ & \geq & 0. \end{array} \tag{4.15}$$

Under the viewpoint of Ben Amor and Desrosiers (2003), the formulation above corresponds to using a 3-piecewise penalty dual cost function. The dual problem adds penalty costs on dual variables when they lie outside of the interval $[\delta_-, \delta_+]$. While du Merle et al. (1999) use a bundle-style algorithm to update the dual center in every iteration, Ben Amor and Desrosiers follow a different approach using a general proximal point algorithm where the center is only moved if no column is found. The method by Ben Amor and Desrosiers (2003) is proved to converge finitely to a dual optimal solution.

This section focuses on the application of a special variant of the model mentioned above in which ε_- and ε_+ are set to infinity. In that case, (4.14) and (4.15) change to the BoxStep model in which the dual variables are kept in a box. The method will converge towards a pair of primal and dual optimal solutions. The algorithm will stop when the current dual point becomes an interior point of the box and all slack and surplus variables are zero. The size of the box could be fixed in advance or changed during the algorithm. At the iterate k, the primal and dual of the restricted master problem in which the box size can be altered are as follows:

$$\begin{array}{ll} \min & {c_{N^k}}^\top x_{N^k} - {\delta_-^k}^\top y_- + {\delta_+^k}^\top y_+ \\ \text{s.t.} & A_{N^k} x_{N^k} - y_- + y_+ = e \\ & x_{N^k}, y_-, y_+ \geq 0 \end{array} \tag{4.16}$$

$$\begin{array}{ll} \max & {u^k}^\top e \\ \text{s.t.} & A_{N^k}^\top u^k \leq c_{N^k} \\ & \delta_-^k \leq u^k \leq \delta_+^k. \end{array} \tag{4.17}$$

At the beginning the box size is kept small so that only locally useful columns can be generated. We will increase the box size gradually in order to satisfy the termination

condition. The BoxStep model will change to the standard model if the box size goes to infinity.

We could also control the box center, creating two general types of the BoxStep method: *sliding BoxStep* and *stationary BoxStep* (same terms as used by Neame, 1999).

Sliding BoxStep

At each iteration the box is centered at the current dual point. The box size is fixed and only increased when the column generation phase stalls. In this case the box size will be incremented by a given constant δB. Therefore, for each iteration, we have

$$\begin{aligned} \delta_-^{k+1} &= u^{*k} - B^l \\ \delta_+^{k+1} &= u^{*k} + B^l \end{aligned} \tag{4.18}$$

If no column is found, we set

$$B^{l+1} = B^l + \delta B. \tag{4.19}$$

Furthermore, if the pricing subproblem stalls in several consecutive iterations, δB should be increased in order to adapt with the hard steps. In experiments presented later on, δB will be multiplied by 2 in such a case.

The method belongs to the generalized stabilized column generation algorithm (Algorithm 4.4.1) suggested by Neame (1999). In the so-called null step, the box center is also moved to the new dual point. In Corollary 4.5.19, the author concludes the finite convergence of the BoxStep method for any sequence of positive box sizes $\{B^{l,i}\}_{i=0}^{\infty} = 1$ such that $\sum_{i=0}^{\infty} B^{l,i} = \infty$. Note, that the author denotes by l the null step iterate, i the serious step iterate. It is also the case in chosen way of updating the box size. Here B^l is the current box size which is initialized to a small value. We will study if there is any effect of δB on the performance of the method. At the beginning, the box center (i.e., u^{*0}) is set to $\mathbf{0}$.

Another variant of the sliding BoxStep method is to keep the box center unchanged until no more column has been priced out and the termination condition has not been met. This can be considered as a mixture of the sliding BoxStep and the stationary BoxStep that will be discussed next. Although there only is a small difference in describing two variants of the sliding BoxStep, they belong to different classes of stabilizing approaches. The second variant comes from the general proximal trust region algorithm of Ben Amor and Desrosiers (2003). Readers can find the convergence analysis of this algorithm for Algorithm A2 in their paper. The only difference is that the authors apply a 5-piecewise linear dual penalty function, instead of a 3-piecewise linear function.

Stationary BoxStep

The stationary BoxStep keeps the center unchanged. In more detail, the two parameters δ_+ and δ_- are controlled directly and independently from the current dual solution.

We will consider an update method different from the one used by du Merle et al. (1999) in which a sequence of δ_+ and δ_- is chosen in advance.

The initial parameter values will be small but large enough to make sure that the box can intersect with the initial cutting planes. The δ-update method chosen for the stationary BoxStep method is:

$$\begin{aligned} \delta_-^{k+1} &= \delta_-^k - \Delta\delta_- \\ \delta_+^{k+1} &= \delta_+^k + \Delta\delta_+. \end{aligned} \tag{4.20}$$

The values δ_+ and δ_- only change when the column generation step is unable to generate more variables. Although the dual center is not changed in the serious step like the method of Ben Amor and Desrosiers (2003) does, the finite convergence proof can be easily seen from the increasing of bounded dual solutions in the serious step, and the increasing of the box size. The change $\Delta\delta$ is likely to have an impact on the BoxStep method. Small values will lead to solving many linear relaxations and pricing subproblems, and otherwise, large values will quickly turn the BoxStep method into the standard column generation method that generates many redundant columns. The idea of increasing $\Delta\delta$ applied in the previous methods is employed here.

In the crew pairing problem considered here, the reduced cost of a variable is $c_j - {u^k}^T A_{\cdot j}$. If we keep u^k small, then few negative reduced cost pairings will be priced out. In other words, the pricing step could be faster. In other papers, the stabilized methods have been discussed with respect to reducing the number of column generation iterations by limiting the zig-zag movement of dual point. Then, as a result, the number of generated columns is also reduced. In addition, this section will show that they can help improve the speed of pricing algorithms considerably.

4.4.2 Computational results

In this section, computational experiments were executed on the same environment used before. The software framework ABACUS has been modified to embed new routines. The previous version of ABACUS does not support the change of objective functions, but this is the case in the stabilized methods. In order to assess the performance of the stabilization methods before applying it to the branch-and-price code, some options of ABACUS have been deactivated, such as *pool separation*, *branch-and-bound*, *primal heuristics*. In the computational experiments, we only focus on solving the root node. Other algorithmic aspects and parameters of the column generation code are the same as mentioned in the previous section.

Since developments of dual variables are the same for all test problems under an individual method presented in Section 4.4.1, the graph of problems "vircpp1" and "vircppl1" is shown in Figures 4.6, 4.7(a), 4.7(b), and 4.9.

Standard column generation method

The computational results of the heuristic enumeration in Table 4.2 shows the poor performance of the standard method shown in the number of column generation iterations ("Lp"). Note, that the enumeration is used in this section. The unstable behavior of dual values may result in a large number of iterations needed to solve the problems. Figure 4.6 visualizes the distance $||u^{*k} - u^*||_2$ between the k-th dual value u^{*k} and the optimal point u^*. The current dual point seems to move far away from the optimal solution in the initial iterations. With the given pricing method, we see that the time needed for the dual variables seems to be more stable lasts until most of the artificial variables have been priced out of the basis. Unlike some other problem classes (e.g., multiple depot vehicle scheduling problem), crew pairing problems are highly constrained by airline rules. It is hard to find an initial feasible solution. Therefore, the addition of artificial variables is required to guarantee the problem feasibility. The tailing-off effect contributes much to the bad performance of the method.

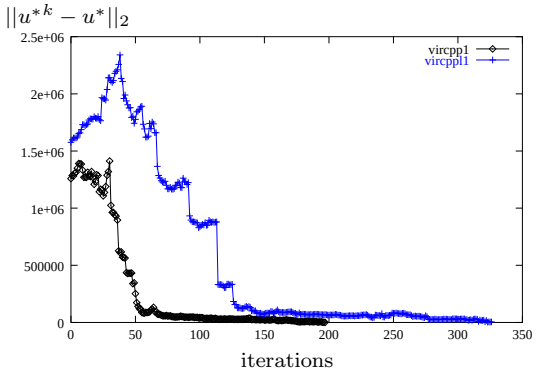


Figure 4.6: Behavior of the dual values in standard method

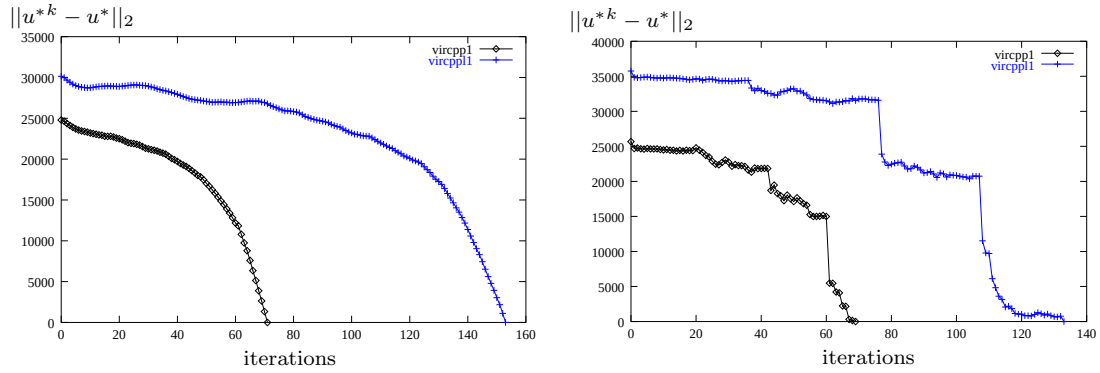
Sliding BoxStep

As mentioned in Section 4.4.1, B^1 is initialized to a suitable small value. Although the initial bounds of the dual variables could be different, doing such a thing has been realized to have a small effect on the computational results. The same value is assigned to all dual components so that the cube $|u| \leq B^1$ intersects with some of the initial set of feasible columns. This guarantees that an enlargement of the box size will not be called right after the first iteration. This idea is applied successfully to solve a wide range of problem sizes (i.e., number of flights). A similar way of initializing the box size will be used for the stationary BoxStep method to be considered in the next sub-section.

Figure 4.7(a) verifies that the aim of stabilizing the dual values has been met. The first variant of the sliding BoxStep method in which the box center will be moved in every iteration makes the dual point move smoothly to the optimal point. The distance $||u^{*k} - u^*||_2$ decreases gradually to zero using the stabilized model. Note that

the initial and final dual points in the stabilized methods could be different from those in the standard method due to their different models.

Under the second sliding BoxStep method in which the box center is not moved if some columns have been generated, the dual point does not move toward the optimal point as steadily as in the first algorithm. The sudden changes in Figure 4.7(b) match with the box size updates. However, bounded regions still keep the dual point moving in a stable manner.



(a) The first variant

(b) The second variant

Figure 4.7: Behavior of the dual values in the sliding BoxStep methods, where $\delta B = 16$ for the first variant, and $\delta B = 128$ for the second variant

Name	n_{B+}	$\underline{z}$	Lp	n^+	$\%_{\text{Pri}}$	$\%_{\text{Lp}}$	t_{Pri}/ite	t_{total}
vncpp1	15	500465	146	799	98.81	1.06	2.89	0:07:07
vncpp2	16	696546	112	645	98.86	1.03	2.95	0:05:34
vncpp3	19	493076	138	676	98.68	1.21	2.82	0:06:34
vncpp4	17	503021	143	636	98.83	1.07	3.17	0:07:39
vircpp1	2	106409	73	516	97.37	2.24	1.11	0:01:23
vircpp2	5	132909	100	620	98.34	1.46	1.86	0:03:09
vircpp3	7	131745	107	532	97.95	1.77	1.11	0:02:01
vircpp4	2	136616	89	647	98.08	1.66	1.55	0:02:21
vircppl1	1	138962	155	1397	98.76	1.14	6.06	0:15:51
vircppl2	2	172947	161	1364	99.38	0.56	10.30	0:27:48
vircppl3	2	179818	149	1287	99.19	0.74	7.77	0:19:27
vircppl4	4	169720	241	1948	99.40	0.56	15.42	1:02:18
sum			1614	11067			57.01	2:41:12

Table 4.4: The first variant of the sliding BoxStep method with $\delta B = 16$

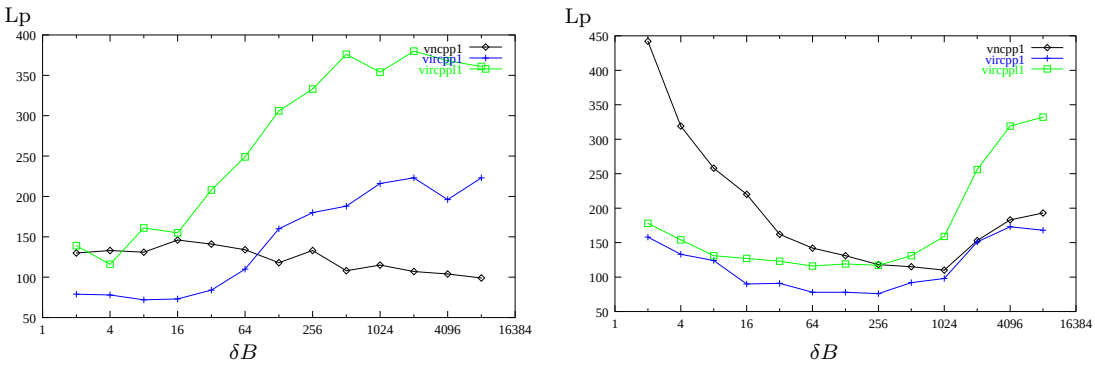
For comparison, a good δB is chosen and the corresponding results are shown in Table 4.4. The method of controlling the dual variables in a box is quite useful. Due to the fact that the path to optimal solutions is more stable, the stabilized method reduces the number of column generation iterations if we look back to Table 4.2. However, the method does not present a good performance with the "vncpp" problems. Frequently the pricing subproblem cannot present any suitable columns (see "n_{B+}"). We can see

Name	n_{B+}	$\underline{z}$	Lp	n^+	$\%_{\text{Pri}}$	$\%_{\text{Lp}}$	$t_{\text{Pri}}/\text{ite}$	t_{total}
vncpp1	39	500465	131	529	98.86	1.03	3.00	0:06:37
vncpp2	39	696546	129	516	98.79	1.08	2.73	0:05:56
vncpp3	39	493076	126	564	98.57	1.28	2.39	0:05:05
vncpp4	40	503021	133	517	98.97	0.95	2.88	0:06:27
vircpp1	7	106409	78	441	96.24	3.26	0.69	0:00:56
vircpp2	10	132909	87	494	97.80	1.85	1.28	0:01:54
vircpp3	10	131745	83	452	98.00	1.70	1.23	0:01:44
vircpp4	8	136616	85	496	96.83	2.70	0.82	0:01:12
vircppl1	6	138962	119	880	96.93	2.78	1.97	0:04:02
vircppl2	9	172947	131	931	98.94	0.96	5.82	0:12:51
vircppl3	9	179818	141	949	98.54	1.34	4.60	0:10:58
vircppl4	9	169720	153	1002	99.41	0.53	9.93	0:25:28
sum			1396	7771			37.32	1:23:10

Table 4.5: The second variant of the sliding BoxStep method with $\delta B = 128$

that the column "$t_{\text{Pri}}/\text{ite}$" of the table indicates that more time is needed to solve the pricing subproblem each time.

Running the second sliding BoxStep on all test problems, we also receive a reduction of iterations (see Table 4.5). The column "Lp" of the table presents half the value in average in comparison with the values of Table 4.2. As a result, the restricted master problem is now supplied with a smaller average number of variables. With two problem sets "vircpp" and "vircppl", the method works well to improve the performance significantly. However, with the set "vncpp", the total time is quite large due to frequent enlargements of the box size ("n_{B+}"). Remember that before increasing the box size, the method has to guarantee there is no column with a negative reduced cost. This process is computationally rather expensive.



(a) First variant

(b) Second variant

Figure 4.8: Number of LPs solved with respect to δB in the sliding BoxStep methods

We come to answer the question raised in Section 4.4.1 about a good initial δB. In Figure 4.8(a), we can see the different behavior of the algorithm according to different problem sets. With the set "vncpp", in which some flights cannot be covered, the change of the number of iterations with respect to δB cannot be seen clearly. By

contrast, the method shows the clear behavior with the set "vircpp" and "vircppl". A small value δB is better than a large one. There are more and more LPs that had to be solved, shown in Figure 4.8(a), when the box size is increased quickly (i.e., larger δB). The small box size in the beginning will guarantee only useful cutting planes to be priced. If we increase this size too fast, the algorithm can return to instability. Note that the box center is not fixed. Therefore, enlarging the box size unreasonably can make the dual point go far away from the optimal point. However, a very small δB also poses a problem by increasing the number of times the box size must be updated. If we update the box size slowly, n_{LP} will increase due to the difficulty of finding a negatived reduced cost column. This was explained before about the poor performance of the method with the problem set "vncpp". Certainly, a good δB depends on the individual problem, especially on the particular cost structure. However, some empirical tests could help us to determine a good δB for the test problems.

Figures 4.8(b) depicts a steady behavior with all problem sets. The box center is also moved in the second method, but more slowly. This avoids the ability of moving too far from an optimal solution in case of the first method. As a result, the second method has a difference range of good δBs. Since the second method updates the box center more slowly, the box enlargement should be increased in order to get a good performance. This is seen clearly in the figures which show that a good δB approximately belongs to the interval [64,1024].

Stationary BoxStep

Figure 4.9 shows that the dual vector is more stable than that of the standard column generation method. However, its graph does not look as smooth as that of the first sliding BoxStep method. The sudden changes in the graph correspond to the times the box size is updated. Similar to the second sliding BoxStep method, the stationary method needs more numbers of iterations because it only enlarges the box size if the pricing problem stalls without any columns.

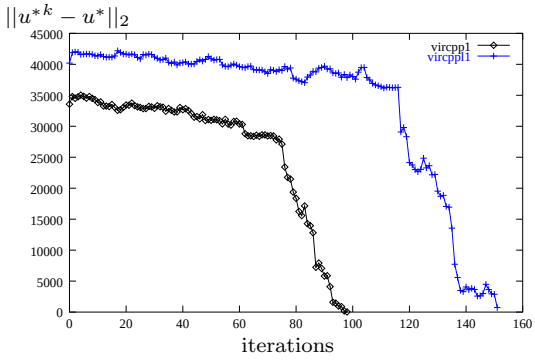


Figure 4.9: Behavior of the dual values in the stationary BoxStep method with the initial $\Delta\delta = 512$

With a good initial value of $\Delta\delta$, this technique has reduced the number of itera-

tions, shown in Table 4.6. This mainly reduces the total elapsed time. Moreover, the method also adds a small number of pairings to the restricted master problem. Useful and strong cutting planes (pairings in the primal problem) have been generated. Apparently, all cutting planes in the dual problem have the form ${A_{.j}}^T u \leq c_j$. Therefore, the distance from that to the origin is $c_j/\sqrt{{A_{.j}}^T e}$ (e is a unit vector). If we gradually enlarge the box centered at the origin, more good planes for the restricted problem are generated. The rather small number of variables generated helps to reduce the time needed to solve the relaxation problems.

Name	$n_{\delta+}$	$\underline{z}$	Lp	n^+	$\%_{\text{Pri}}$	$\%_{\text{Lp}}$	t_{Pri}/ite	t_{total}
vncpp1	7	500465	119	839	97.31	2.42	1.57	0:03:12
vncpp2	7	696546	125	830	97.87	1.96	2.06	0:04:23
vncpp3	7	493076	142	939	97.47	2.38	2.09	0:05:04
vncpp4	7	503021	116	792	96.58	3.15	1.22	0:02:27
vircpp1	2	106409	101	812	92.43	6.81	0.50	0:00:55
vircpp2	3	132909	109	773	95.48	4.10	0.81	0:01:33
vircpp3	3	131745	98	777	91.79	7.57	0.41	0:00:44
vircpp4	3	136616	114	813	95.10	4.47	0.67	0:01:20
vircppl1	2	138962	154	1375	96.41	3.33	2.35	0:06:16
vircppl2	3	172947	164	1334	98.61	1.29	5.72	0:15:52
vircppl3	3	179818	160	1359	97.93	1.94	4.30	0:11:43
vircppl4	3	169720	198	1588	98.86	1.06	7.65	0:25:32
sum			1600	12231			29.28	1:19:01

Table 4.6: The stationary BoxStep method with the initial $\Delta\delta = 512$

One interesting point is that the pricing per iteration is quite small in comparison with the standard method and other stabilized methods. This means that pricing in the stationary approach is faster due to the limitation of the box size and the fixing of the box center, which is verified by the pricing time per iteration in Tables 4.4 and 4.6. This is explained by recalling the cost structure of the crew pairing problems mentioned before. With small absolute values of dual variables, the number of possibilities for appending a flight to a negative reduced cost pairing is reduced as well. This limits the combinatorial explosion in the pricing problem. We also see a small pricing time per iteration in the second sliding BoxStep method in Table 4.5 because this method can be considered as a combination of the first sliding BoxStep and the stationary BoxStep.

The parameter $\Delta\delta$ plays a very important role in the stabilized technique, as shown in Figure 4.10. Different $\Delta\delta$ present different computation performance values. On the left side of the figure, small values of $\Delta\delta$ will make the column generation step stall many times. This leads to the fact that many LP problems and pricing subproblems have to be solved. The result is possibly a high computation time. Like the previous BoxStep methods, we do not easily obtain more variables if we allow the dual point to move within a very small region. The bad performance also occurs in case of large values of $\Delta\delta$, but due to a different reason. When the box size increases quickly, the stationary BoxStep method soon becomes the standard column generation method.

Choosing a good $\Delta\delta$ is crucial in reducing the total computation time. In the class of the Vietnam Airlines crew pairing problems, the cost of a pairing will be less than 10080 (i.e., the number of minutes in a week). Hence the distance from a cutting

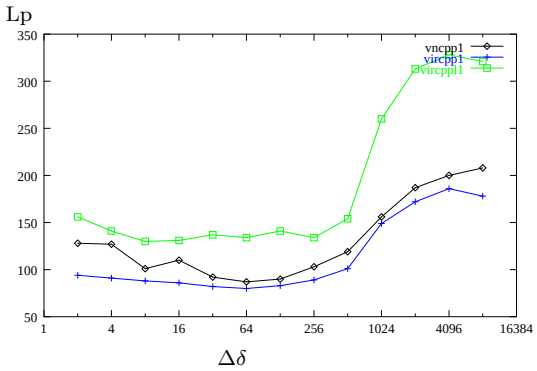


Figure 4.10: Number of LPs with respect to different $\Delta\delta$ in the stationary BoxStep method

plane to the origin is less than $10080/\sqrt{2} \simeq 7128$ (a pairing has at least 2 flights). It is unreasonable to choose $\Delta\delta$ larger than this number because the dual restriction mechanism will work inefficiently in that case. The smallest is about 31 with respect to the considered airline rules. Within the interval $[31, 7128]$, the interval [64,128] seems to contain the best values. Up to now, any mathematical relationship between the best $\Delta\delta$ and the properties of the considered crew pairing problem (cost structure, flight regulations, flight network, etc.) has not been found yet. The best parameters are only chosen based upon empirical tests.

Branch-and-price with the stabilized methods

In this sub-section, the application of the mentioned stabilized methods in the branch-and-price code will be presented. Due to performance reasons, only the second variant of the sliding BoxStep, the stationary BoxStep are implemented into the framework. The first sliding BoxStep does not work well on the test problems.

Observing the development of dual points in the non-root node, their path is seen to be quite stable and only a few column generation iterations are needed to reach optimality. This can be explained by the fact that the current set of pairings is quite "strong" to present a good initial dual point. However, there are many possible dual optimal solutions. This means the solution of the stabilized methods is possibly different from that of the standard method. If we use the stabilized model for higher nodes (e.g., root node) and the standard model for lower nodes (e.g., non-root node), the lower nodes might need more iterations than usual. This is because starting from a set of pairings created by a stabilized model it is harder to reach optimality in the standard model. The first application of the stabilized methods follows the idea of using them in the root node but with a poor performance.

After that, the stabilized methods is employed in every branch-and-bound nodes. In implementation, the stabilized variables are included into the model of the root node and will be kept there for their child nodes. After branching, the associated problem of a node is likely to be infeasible. Therefore, the artificial variables are always kept

in the model of every node. Remember that, for the root node, the sliding BoxStep sets the initial box center to 0. But we can reasonably reuse the final dual solution of a parent node as the initial dual box center of its child nodes. This is also performed in case of the stationary BoxStep.

The complete computational results are displayed in Table 4.7. With the sets "vncpp" and "vircpp", the code with the sliding BoxStep works well in several cases and badly in others. It is hard to make any claim on the total number of LPs in comparison with the standard approach. However, the total number of generated variables in many cases is declined. This explains why we have a smaller percentage of time involved in solving linear programs. We receive a similar result with the stationary BoxStep. Note, that we cannot guarantee that the stabilized methods will improve the performance of branch-and-price runs in all cases.

In the table, we see there are very few improvements by the local heuristics. Note that these heuristics depend on the replacement of bad pairings by better ones. But with the stabilized methods, most of generated pairings are quite good. Another reason for that phenomenon is that the methods have found very few feasible solutions. Consequently, the same happens with the reduced cost fixing. This is a disadvantage of the stabilized methods. Due to the existence of the stabilizing variables, it is harder for the restricted master problem to be integer feasible. Normally, solving only the root node of a problem, the standard branch-and-price code has introduced several feasible solutions.

Meanwhile, the methods solve the large problems in the set "vircppl" more quickly. The stabilized devices have demonstrated their ability to deal with a huge number of variables in such problems. The stabilized codes now find the optimal solution within two hours. The branch-and-price with the stabilized column generation methods is useful for large-scale problems.

Decomposition is a powerful method to solve many classes of combinatorial optimization and column generation is often a choice for linear relaxation. But one of its main problems is the instability of dual variables. The zig-zag movement of the dual point leads to a large number of column generation iterations, and consequently, generates a huge number of variables which could be inactive in the final solution with a high probability. This also happens in solving crew pairing problems which are of our interest. With the three stabilizing approaches, we have reported the efficiency of our implementation in solving certain crew pairing problems. Controlling the dual vector has been proven helpful to reduce the number of column generation iterations and variables priced out. The sliding and stationary BoxStep methods make the dual point move more smoothly to the optimal point, leading to a considerably improved performance with respect to the standard method. The choice of the parameters δB and $\Delta\delta$ turned out to be crucial in the methods. The computation time can be reduced by nearly a factor of three with a careful selection of these parameters. There is still an open question of the relationship between the parameters and our crew pairing problems.

In the next chapters, the thesis will follow a different direction to use the computing

Name	$\overline{z}$	B&B	Lp	n^+	n^-	heuristics			time						
						2-opt	scroll	merge	$\%_{\mathrm{Pre}}$	$\%_{\mathrm{Pri}}$	$\%_{\mathrm{Heu}}$	$\%_{\mathrm{Bra}}$	$\%_{\mathrm{Lp}}$	t_{Pri}/ite	t_{total}
The sliding BoxStep (second variant), $\delta B = 128$															
vn320	64465	1	93	466	0	0	0	0	0.02	99.64	0.00	0.00	0.28	6.71	0:10:26
vn321	38870	1	40	104	0	0	0	0	0.21	94.57	0.00	0.00	4.15	0.21	0:00:09
vnAT7	22260	1	45	272	0	0	0	3	0.27	85.79	0.00	0.00	10.42	0.13	0:00:07
vncpp1	500943	9	163	561	2	0	0	0	0.01	97.61	0.00	0.02	1.03	3.71	0:10:19
vncpp2	697445	9	172	568	9	0	0	0	0.01	97.46	0.00	0.02	1.13	3.51	0:10:19
vncpp3	494154	223	1217	1784	1	0	0	0	0.02	94.37	0.00	0.06	1.51	4.58	1:38:26
vncpp4	506006	163	1073	1517	1	0	0	0	0.01	95.88	0.00	0.04	1.07	5.50	1:42:36
vircpp1	107429	41	282	708	0	0	0	0	0.02	93.48	0.00	0.05	1.42	2.99	0:15:02
vircpp2	134421	443	2065	2759	0	0	0	0	0.02	91.42	0.00	0.06	1.50	3.98	2:29:45
vircpp3	135116	19	313	763	0	0	0	0	0.01	97.63	0.00	0.01	0.76	4.34	0:23:13
vircpp4	137165	31	314	889	0	1	0	0	0.02	95.51	0.00	0.03	1.31	3.17	0:17:21
vircppl1	139344	45	332	1225	0	0	0	0	0.01	94.15	0.00	0.02	1.06	12.55	1:13:46
vircppl2	173538	23	276	1199	0	0	0	0	0.01	97.34	0.00	0.01	0.59	19.72	1:33:12
vircppl3	179818	1	141	949	0	0	0	0	0.03	98.50	0.00	0.00	1.36	4.52	0:10:47
vircppl4	169852	39	197	1062	0	0	0	0	0.00	94.86	0.00	0.01	0.61	19.14	1:06:16
The stationary BoxStep, $\Delta\delta = 64$															
vn320	64465	1	92	496	0	3	0	3	0.01	99.42	0.00	0.00	0.46	3.87	0:05:58
vn321	38870	1	33	70	0	0	0	3	0.32	93.53	0.00	0.00	4.85	0.17	0:00:06
vnAT7	22260	1	42	242	0	0	0	3	0.08	92.46	0.00	0.00	5.43	0.26	0:00:12
vncpp1	500943	29	151	609	0	0	0	0	0.00	93.74	0.00	0.06	1.34	3.25	0:08:44
vncpp2	697445	9	155	663	9	0	0	0	0.01	96.83	0.00	0.03	1.53	3.37	0:08:59
vncpp3	494154	369	1572	2362	2	0	0	0	0.02	93.50	0.00	0.07	1.63	5.30	2:28:38
vncpp4	506006	297	1566	2180	56	0	0	0	0.05	94.64	0.00	0.05	1.44	5.04	2:18:54
vircpp1	107429	41	259	743	1	0	0	0	0.01	92.16	0.00	0.05	1.87	2.85	0:13:20
vircpp2	134421	209	906	1447	2	0	0	0	0.02	89.81	0.00	0.07	1.99	3.55	0:59:41
vircpp3	135116	13	250	773	0	0	0	0	0.00	96.82	0.00	0.02	1.17	2.95	0:12:41
vircpp4	137165	19	209	767	0	0	0	0	0.00	96.20	0.00	0.03	1.27	3.72	0:13:29
vircppl1	139344	23	282	1349	0	0	0	0	0.00	97.07	0.00	0.01	0.78	17.58	1:25:07
vircppl2	173538	21	267	1303	0	0	0	0	0.02	97.07	0.00	0.01	0.72	17.68	1:21:02
vircppl3	179818	1	135	1045	0	0	0	0	0.00	97.92	0.00	0.00	1.92	3.66	0:08:25
vircppl4	169852	11	184	1171	0	0	0	0	0.01	97.81	0.00	0.01	0.59	15.95	0:50:01

Table 4.7: Computational results of the branch-and-price code with the stabilized methods

power resource of parallel systems to solve the crew pairing problems. The main objective is still to speed up the solution process.

Chapter 5

Parallel ABACUS

5.1 Aspects of Parallelization

It is quite obvious that a parallel approach is a good choice for difficult problems which cannot be solved in a reasonable computation time by a sequential implementation (see Chapter 3). This approach especially makes sense in solving combinatorial optimization problems, such as our crew pairing problem. In the branch-and-cut approach for crew pairing problems discussed in Chapter 3, there are difficulties (computation time, memory) that arise in the process of solving test problems. In order to have a "good" design of a parallel solver, we will focus on its related issues. Note that, although there are various aspects of parallelization in general, the thesis mainly aims at reducing the computation time. Other aspects are less considered. Readers interested in them are referred to Chen and Ferris (1999), Eckstein et al. (2000), Ralphs et al. (2003) for related works. Besides contributing parallel implementations, they also give us general reviews on the design of a parallel branch-and-cut-and-price solver.

5.1.1 Communication library

ABACUS has been parallelized to work on clusters of computers by Böhm (1999). Distributed memory systems are also the targeted environments. They consist of a communication network connecting a group of processors having their own and separated memory. In the old parallel ABACUS, the communication among processors is performed by the Adaptive Communication Environment (ACE) library (see Schmidt, 1998) which is a good library in terms of flexibility, portability, and thread-safety. It is quite good in developing applications on distributed systems because it supplies a diversity of functions and classes for concurrency and synchronization control, interprocess communication, event de-multiplexing, etc. In other words, the library provides a good programming environment on the layer of distributed object computing middle-ware. However, it does not care much on performance issues. The high-overhead TCP/IP of operating system is its communication sub-layer. Therefore, it should not be used for highly computing-demanding applications. Moreover, it is not a standard and well-

supported communication library on many parallel computers nowadays and in future. Therefore, it will likely be an obstacle on the progress to use parallel ABACUS as a framework for discrete optimization.

An effort should be made to migrate it to high performance systems using the message passing as a main communication model. *Message Passing Interface* (MPI) (see Message Passing Interface Forum, 1995) will be chosen not only because it is the most famous message passing standard, but also it is implemented on many parallel computers. There is another famous message passing library called *Parallel Virtual Machine* (PVM) which has been developed at Oak Ridge National Laboratory, USA. There are many books (e.g., Geist et al. (1994)) about this library and it is also mentioned within books on parallel computing. However, MPI is preferred to PVM because MPI often delivers a better communication performance on parallel PC-based computers.

The design of the ACE-based parallel ABACUS strongly bases on the multi-threading support of ACE in which each thread is assigned a specific task and is programmed independently. They run simultaneously to execute the branch-and-cut-and-price kernel, to share bound information, branch-and-bound nodes and other necessary data. Meanwhile, standard MPI 1.0 has not supported multi-threading yet. Version 2.0 of MPI overcomes the shortcoming of the predecessor and, furthermore, supplies features which can be used to modify the ACE-based library without a big change of design. Nevertheless, none of the implementations of this version are available. A new design of the communication has to be exploited in order to efficiently use current MPI libraries and prepare for further extensions.

5.1.2 Task granularity

In this section, we will consider different levels of parallelization in order to get a good performance. In general, a small unit of work results in much communication cost. By contrast, a large unit of work presents possible a bad load balancing. In any branch-and-cut-and-price solver, there are mainly three computationally intensive parts:

Linear relaxation

As seen in Chapter 3, the sequential program often spends a major portion of time in this phase for difficult problems (e.g., "aa01", "aa04"). Therefore, it is a good target for parallelization. As mentioned in Chapter 1, there are two main methods in linear programming: the simplex method (including its variants) and interior point methods. There are some efforts to parallelize them. Since the simplex method is quite effective in a branch-and-cut framework, we will consider it primarily in the thesis. Yarmish (2001) investigates many aspects of a distributed implementation of the simplex method. In his work, readers can easily see a limited number of processors on which the parallel simplex solver performes best. The simplex method strongly depends

on the movement from vertex to vertex. We can only benefit from the parallelization of each iteration, which is not much useful. Besides that, the experiments in Yarmish's work also underline this finding. The limitation is a critical point that has a strong impact on the extensibility of his parallel solver. More information on the parallel simplex method can be seen in that thesis.

Coleman et al. (1996) implement a parallel interior point solver for linear programming. The parallel part is mainly for sparse Cholesky factorization. Their solver is designed to solve very large size Netlib LP test problems. This is due to the fact that the factored matrix is distributed among processors. The distribution is a good approach for dense and large matrices which are impossible to be stored on the memory of one processor. Although interior point methods are of a polynomial computation time, it is still not easy to "warm start" after violated cutting planes have been added into model. Therefore, we will not make any further study on this subject.

Moreover, with test crew pairing problems (e.g, "nw04") which require much of the separation time, the approach of only parallelizing the linear solver cannot give a good speedup. Amdahl's law gives us a formulation to compute the theoretical speedup in which there is a strict sequential part B (in percentage) of an algorithm. In more details,

$$S_p = \frac{p}{B \times p + 1 - B}, \tag{5.1}$$

where p is the number of processors. For problems involving more than 50 percent of the total computation time for separation (and/or preprocessing, heuristics), a parallel linear relaxation is not enough to obtain a good speedup. In other words, the maximum speedup is limited to $1/B$. If we want a good speedup on quite a large number of processors, we should think about different or at least additional ideas.

Cutting plane separation

All cutting planes which are used in the sequential implementation are globally valid. Time to separate them is small for most of the test problems. It is not necessary and effective to parallelize the separation. Fast separation algorithms such as row-lifting, greedy, GLS, are better to be implemented in sequential. We will see in Chapter 6 the variable pricing is more worthy to be parallelized. One thing which can come into vision is how to store these global inequalities. Several parallel designs keep the global objects in a center where they will be delivered to processors on demand. For example, this approach is used by PICO, FATCOM, COIN/BCP, and SYMPHONY. This centralized idea is also applied to store open nodes on these parallel framework. Storing all data on a master makes the design easier to be implemented. However, there are some problems likely to appear. The first is that a bottleneck can arise at the master if the number of processors increases, limiting the scalability of the design. In order to avoid this problem, the idea of worker-hub-manager framework has been suggested. The pool of processors now will be partitioned into clusters which have a sufficient number of processors and are controlled by a hub. In this way, there are a

limited number of processors communicating with their hub which, in turn, exchanges data with a unique manager. Another problem is that we cannot use the master for solving nodes if it is idle. Although it is possible to implement that idea, it will turn to be a complicated task.

Preprocessing, heuristics

Similar to the two aspects above, the parallelization of just these steps is not preferred. We can solve each part in parallel in order to reduce the effect of Amdahl's law. However, time to gather data and transfer data back to computing processors can dominate the overall computation time. Remember that we aim to design a parallel solver for a loosely coupled system in which the memory is distributed and the network connection often has a low bandwidth and a high latency.

Some researchers use a high synchronization approach for parallelization implementations. This means all processors solve each phase of a typical branch-and-cut code together. One example of this approach is the work of Linderoth et al. (2001) which also attacks large scale set partitioning problems as the thesis will show later. The advantage of this approach is that the branch-and-bound tree is the same for both sequential and parallel runs with a high probability. In that work, the lower communication layer also uses a library (PVM) for distributed memory systems, and its implementation has been tested on a MPP (IBM RS/6000 Model 390). In their computational results, the speedup is not good even with a small number of processors (less than 16 processors). Time reduction in one phase does not guarantee the reduction of total time. Certainly, despite the same branch-and-cut settings, any sophisticated branch-and-cut framework can give different computation times under slightly different environments. Therefore, it is quite difficult to compare parallel solvers. However, we prefer an implementation introducing a good speedup even with a quite large number of processors.

With the remarks above, choosing a single branch-and-bound node as a unit of work is the most suitable for a parallel branch-and-bound solver. We will consider how to solve branch-and-bound nodes on processors and how to store them across a network.

5.1.3 Pool of nodes and load balancing

Most parallel branch-and-bound solvers employ the idea of storing open nodes in one (or few) processors. Furthermore, these processors are only responsible for performing management tasks. The central processor follows a node selection strategy to pick nodes from the pool and distributes them to computing processors under a job delivering strategy. The approach can be seen in many new parallel solvers, such as SYMPHONY, COIN/BCP, FATCOP.

However, our parallel ABACUS uses a different way of storing open nodes. Every processor has its own pool of nodes. We can call them local pools. The branch-and-cut core on a processor will select the next candidate to be processed from its local pool. In addition to the local pool, processors have a mechanism to share their open nodes with

others. A good sharing should pay attention to the load balance and other information closely related to the branch-and-cut solver.

5.1.4 Object oriented design

ABACUS is designed as an object oriented framework which supports developing branch-and-cut-and-price based applications. It consists of a collection of classes. Most of them are designed as a black-box which hides the algorithmic structures of a typical and complicated branch-and-cut-and-price algorithm from the user. Besides that, ABACUS requires that some of its abstract classes must be derived to create problem specific classes. They will help ABACUS to know how to build a combinatorial problem from the user data structure and how to solve that problem under the branch-and-cut-and-price principle. There is a wide range of classes and their functions which can be redefined to satisfy a variety of user requirements. A small example as the TSP solver discussed in the ABACUS guide shows that ABACUS is an easy-to-use framework. Moreover, with ABACUS, a simple code can be improved to solve difficult real-world problems. This `C++` framework satisfies its own design criteria: flexibility, extendibility, easy-to-use, functionality and portability. Furthermore, ABACUS also tries to guarantee the aspect of efficiency by considering carefully its internal design. Redundant parts are removed as much as possible with the other design criteria above being only impacted slightly. As as result, ABACUS is widely used in the combinatorial optimization community. More details can be found in the work of Thienel (1997).

The design of the parallel ABACUS should not break the good properties of the sequential version. Moreover, the design also aims at users who have little experience in parallel programming. This means that users will only be required to add few simple classes and functions to their sequential codes in order to have a parallel version. Most aspects of parallel programming should be invisible from a user's viewpoint.

The design of the parallel ABACUS is visualized in Figure 5.1. A class, named `ABA_PARMASTER`, is added to be responsible for communication among processors. As mentioned in Section 5.1.3, each processor has its own local pool of open nodes. In more detail, the pool of the sequential ABACUS is still the pool in the parallel version. The main difference is that the local pool is accessed not only by the ABACUS kernel (for computation on the local processor), but also by a balancer which will send open nodes to other processors as requested. The notification server will share information with other processors in order to keep the global state information in `ABA_PARMASTER` up-to-date.

`ABA_PARMASTER` keeps global information of parallel runs. It also responds to notification messages from other processors, such as bound changes, new numbers of open nodes, termination checking. ACE provides a capability to implement easily event handlers for these messages. The sequential ABACUS has been changed to send or broadcast the notification messages when the state of the ABACUS changes. The balancer and notify server run concurrently as threads along with the ABACUS kernel. This will be a painful problem in the process of replacing ACE by MPI. We will solve

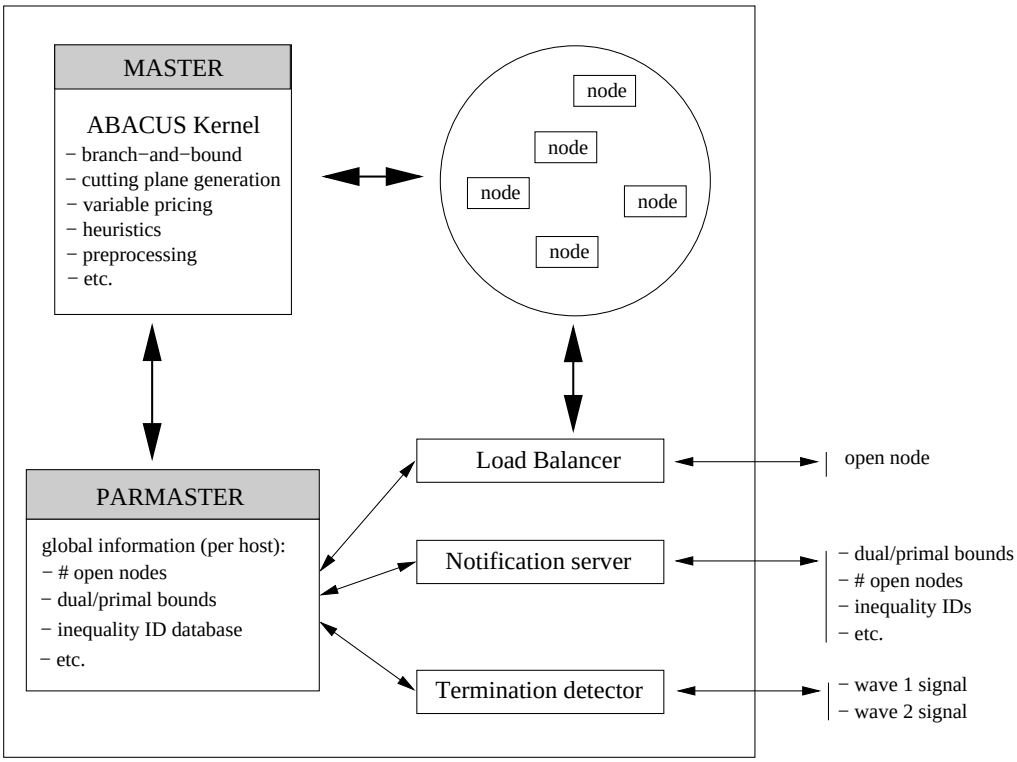


Figure 5.1: Architecture of the parallel ABACUS on a processor

this problem in Section 5.2. Remember that users do not have to pay attention to three added classes. Users are only required to provide some additional lines of code described in the next section.

5.1.5 Object serialization

MPI has the ability of using user memory space as communication buffer. This means it will copy data inside the buffer directly to other processors. By doing so, a lot of memory copies are reduced. However, this approach presents some difficulties. First it creates troubles in designing a parallel framework. Users are possibly required to call communication functions by themselves because the framework cannot know in advance which specific data is needed to be exchanged among processors. This is in contrast to one of our design criteria: hiding as many the parallel parts of ABACUS as possible. Another problem links to the design of communication using MPI. If we use the user memory for communication, it cannot be released or reused until the associated communication function is completed. In a single-threaded environment, this will block other funtions. It is worse if the parallel code is not designed carefully, leading to deadlocks.

A solution is to copy the user data into a temporary communication space. Such a process of copying is called *serialization*. Users have to redefine virtual functions to

copy necessary data of a class into a message which will be transferred by the communication library and to recreate the object from the received data. Classes which possess such functions are `ABA_SUB`, `ABA_CONSTRAINT`, `ABA_VARIABLE`, and `ABA_BRANCHRULE`. The function `pack()` is called to put data members into a message. A message constructor will rebuild the object from a message. The real communication is performed by the parallel ABACUS kernel. To pack and unpack data members, users must call functions provided by the parallel framework. They are supplied to deal with all primitive data types and arrays of them. Since only users know how to deal with a received message, they are required to provide an additional function `unpackConVar(...,Id)` and `unpackSub()` to the class `ABA_MASTER`. `Id` determines which user variable and constraint will be recreated. Therefore, it must be unique for each object and returned by the function `classId()` of the object. A simple example of how to parallelize a TSP solver is shown in Böhm (1999).

5.2 New Communication Design

The ACE-based parallel ABACUS is not designed as a master/slave architecture to avoid bottlenecks. Instead, it performs a branch-and-bound tree decomposition in parallel. Each processor runs a sequential ABACUS kernel to solve its list of open nodes. The main difference between the sequential and parallel libraries relates to the sharing of crucial data of the branch-and-bound framework. To do the sharing, the library employs ACE to create several threads running concurrently. These threads access the ABACUS kernel to get data, and exchange those data with corresponding threads on remote processors which, in turn, put the data to the ABACUS kernel on the remote sides. On one processor, if a thread is suspended by the operating system, there is no influence on the progress of other threads. This means that the blocking communication operations in ACE library are used with a small effect on the performance of the kernel.

As discussed in Section 5.1.1, a new design of the communication is required for the current MPI standards which support only one thread. The data exchanges will be performed by periodically calling certain communication functions. The idea is quite similar to PICO. However, it is not necessary to design a virtual thread scheduling layer as in PICO. Instead, the new design will put those functions directly into the sequential ABACUS code. This is also different from the ACE-based design due to the single thread nature. It is critical to decide where to put those functions in the branch-and-bound framework. In order to get a good performance for the new library, we have to consider carefully the following two main factors:

(i) the time between two consecutive communication calls,

(ii) the time for a communication call.

There are several kinds of data to be exchanged among processors and their rates of changes are certainly different. The position of their update functions within the

ABACUS code are hence decided individually. This is also the answer to the factor (i). For (ii), the blocking communication cannot be used because the communication time will have a strong impact on computational performance. This is obvious when a receive side is waiting for an incoming message, and all other computations and communications will be suspended. An alternative mechanism is to use nonblocking communication which often leads to better performance by overlapping communication and computation. It is also a basic communication mode not only supported by all message passing standards, but also implemented in all MPI libraries. Using nonblocking functions, we can obtain advantage from some lower level communication networks which can exchange data without any impact on the computing power of processors. In summary, the asynchronous message passing approach by nonblocking functions avoids inherent synchronization issues, which can create long delays in a multiprocessor environment. However, it is hard to design a system using nonblocking communication and one needs much tuning in order to get a good performance. After launching a nonblocking communication, the flow of execution must decide when to check its completion while not breaking the main computation into fragments. In order to apply this mechanism, we implement two functions for each kind of exchanged data: one for the send side and another for the receive side. Their structures are shown in Functions 5.2.2 and 5.2.1.

Function 5.2.1 Nonblocking receive function

```
1: if (flag[p] == true) then
2:    <probe any incoming message from p>
3:    if (there is a message) then
4:       <start a nonblocking receive>
5:       flag[p] ← false
6:    end if
7: else
8:    <check the completion of the nonblocking receive>
9:    if (the receive is completed) then
10:      <call a corresponding unpacking function>
11:      <transfer received data into ABACUS kernel>
12:      flag[p] ← true
13:   end if
14: end if
```

Remember that p is a remote processor. And the broadcast will be performed by a loop through all processors, except the local one. In Function 5.2.1, the variable $flag[p]$ operates as a state variable for receiving messages from a given processor i. If it is `true` it means that no nonblocking receive for p has been submitted. If so, the receive function probes for any incoming message and launches a nonblocking receive if some exists. Right after launching, the state variable $flag[p]$ will be changed to `false` in order to guide the function to check the completion of the receive. Then,

a corresponding unpacking task is performed depending on the kind of received data. Two phases in the receive function are also essential in order to adapt to different message sizes.

Function 5.2.2 Nonblocking send function

```
1: <check the completion of the previous send to p>
2: if (the previous send is completed) then
3:    <pack the data to be sent into a message>
4:    <start a nonblocking send of the message>
5: end if
```

The send function is more simple than the receive one. In a similar way, an MPI nonblocking function is also used. Another important point is that the previous send must finished before the next one can be started. This is reasonable with all kinds of data though it may delay the information update to remote processors. However, for certain important information which must be up-to-date, cancelling a sending can cause the whole parallel search to show strange phenomena. For example, the lack of updating the dual bound of other processors makes the local processor give a wrong decision on load balancing. Therefore, in the real implementation we use a more complicated flowchart for those kind of data to guarantee that all necessary messages will be sent. In broadcasting such a kind of data, the loop through all remote processors will be stopped immediately if a send cannot be performed. Then, a flag is set in order to restore the function from the stopping point.

In the ACE design, mutex operations must be used to solve conflicts on accessing objects shared among threads (e.g., the list of open nodes, constraint/variable pools). Therefore, besides the communication time, the overhead of the ACE-based library is mainly from those operations and thread switching. It is quite clear that the performance of the new MPI design will be affected by the polling for completions of the nonblocking operations. In the case no communication is needed, the performance expense of the functions is quite small, only coming from conditional statements. The position of update functions for each kind of data will be discussed separately in order to reduce the overhead of the functions.

Before going further, readers should remember that generally there are two main loops in the branch-and-cut-and-price framework. `ABA_MASTER` controls the outer loop (see Algorithm 5.2.3) of the branch-and-bound algorithm which selects nodes one-by-one and optimizes them with a cutting plane or column generation algorithm. That is the inner loop (see Algorithm 5.2.4) which belongs to `ABA_SUB`. Since nonblocking send and receive functions will be inserted directly into the code, blocks in Figure 5.1 do not mean independent threads any more. Instead, they can be thought of as functional blocks. The block `NOTIFYSERVER` is dropped out of the new design because notification events will be handled by `PARMASTER`.

Algorithm 5.2.3 Branch-and-bound loop with communication functions

```
 1: while (true) do
 2:    s ← next open node
 3:    if ( s != NULL) then
 4:       <optimize s>
 5:       if (there is an error) then
 6:          break
 7:       end if
 8:       if (s is fathomed) then
 9:          <delete s>
10:       end if
11:    else
12:       <start the termination check>
13:       if (termination condition is satisfied) then
14:          break
15:       end if
16:       <update the global dual bound>
17:       <update the sets of constraint/variable IDs>
18:    end if
19:    <update the global primal bound>
20:    <update new numbers of open nodes>
21:    <call a function to balance the number of open nodes>
22: end while
```

Bounds and number of nodes

Functions exchanging primal bounds could be inserted into the outer loop (line 19 of Algorithm 5.2.3). However, the primal bound receive function is coded within the inner loop at line 4 of Algorithm 5.2.4 in order to get the current node fathomed as quickly as possible. Obviously, if a new primal bound is found by a processor, it will be sent immediately to other processors. Hence, the function to send the bound is called immediately after the discovery. A new dual bound on a processor is broadcasted as soon as the those functions for primal bound is finished. However, the update of the dual bound among processors should be performed more regularly. It is the reason why its update functions are placed within the cutting loop. However, the functions are also called when the processor is idle (line 16 of Algorithm 5.2.3). Normally, the global dual bound should be better (e.g., increasing in a minimization problem). However, in parallel runs, the global best dual bound can get worse. This does not happen in a sequential execution but could occur in parallel. When a processor has finished sending a node and its corresponding remote processor has not received the node, the ABACUS layer has no information on the node. Thus, the bound of this node is not known by any processor, possibly leading to a better global dual bound. After receiving the node, the global dual bound will be updated back to the worse value (i.e., the dual bound of

Algorithm 5.2.4 Cutting and pricing loop with communication functions

```
1: while (true) do
2:    <update the global dual bound>
3:    <update the sets of constraint/variable IDs>
4:    <update the global primal bound>
5:    <call the cutting plane phase of the sequential ABACUS>
6: end while
```

the node just received).

Two functions for the number of open nodes are placed within the outer loop because this counter only changes in this level (see Algorithm 5.2.3). Whenever a node is removed from the local pool, the processor will broadcast its new state to all others. The number of open nodes must be sent successfully, especially when there is no open node. This is important for load balancing.

Open branch-and-bound nodes

Different from the bound exchanges, if the node send function finds out that the previous send has not been completed yet, it will not discard the send. Instead, it puts the node back to the local pool. We will see more details in Section 5.3 which will discuss a new load balancing mechanism.

Similar to the ACE-based library, the new library sends only identification numbers (IDs) of constraints and variables. Each processor has a database storing sets of identifications of constraints and variables which are currently available on other processors. Using this database, a sender could decide which constraints and variables should be delivered along with the node. Note, that constraints/variables of original integer formulations certainly do not have to be sent because they always exist on every processor during the optimization startup. Since those identification databases sometimes are not synchronized correctly among processors, it is possible that a sender can send more objects than needed. In such a case, there occurs no error to the node which is recreated on the receiver because the receiver will discard redundant objects.

In order to reduce the number of redundant constraints or variables sent, it is essential to update the identification databases. A broadcast of newly generated identification numbers is executed at the end of a separation or pricing. Since in each separation or pricing there is a number of new constraints or variables, they are collected into a package to be sent together. The receive function of identification numbers is presented in both Algorithms 5.2.3 and 5.2.4 because the identification databases should be up-to-date.

5.2.1 Termination detection

It is clear that the execution cannot quit the branch-and-bound loop although there is no open node in the local pool. The processor must wait for unsolved nodes from

other processors or a termination signal emitted from a given processor. Moreover, it is possible that all processors are idle but some messages have not been delivered yet. They can cause certain processors to restart computation. Therefore, we need a stable algorithm to detect the termination condition when all processors are idle and there is no more message in transit. The termination detection is performed by the *four counter method* (see Mattern, 1987) as following.

Let $s_p(t)$ and $r_p(t)$ be the numbers of messages sent and received respectively by the processor p at the global time instant t. The total numbers of messages sent and received at the global time instant t are

$$\begin{array}{l} S(t) = \sum_p s_p(t) \\ R(t) = \sum_p r_p(t). \end{array} \tag{5.2}$$

Unfortunately, the global time instant t cannot be obtained. Instead, we can only have values:

$$\begin{array}{l} S^* = \sum_i s_i(t_i) \\ R^* = \sum_i r_i(t_i). \end{array} \tag{5.3}$$

In order to terminate the global computation safely, all processors are idle and $S(t) = R(t)$ (i.e, all sent messages have been received). To ensure that, the four counter method implements counters $s_p(t)$ and $r_p(t)$ on each processor. One dedicated idle processor starts a control wave to collect these counters from all others. If it sees $S^* = R^*$ (we denote them by S_1^*, R_1^*), it will start the second wave also to obtain values of the counters. We denote the new sums by S_2^* and R_2^*. If four numbers are equal, the computation is terminated.

Let t_1 be the time the initiator starts the first wave, t_2 be the time it receives the counters, t_3 be the time it starts the second wave, and t_4 the time it receives the counters of the second wave. We the have the following remarks:

(i) if $t \leq t'$, $s_i(t) \leq s_i(t')$, and $r_i(t) \leq r_i(t')$.

(ii) if $t \leq t'$, $S(t) \leq S(t')$ and $R(t) \leq R(t')$ (due to (i)).

(iii) Because we can only accumulate counters at different time instants, we should find a relation between S and S^*, R and R^*. Since the counters on processors can increase after replying to the initiator, we have $S_1^* \leq S(t_2)$, and $R_1^* \leq R(t_2)$. Deducing in a same way, $S(t_3) \leq S_2^*$, and $R(t_3) \leq R_2^*$.

We have

$$\begin{array}{lll} R_1^* = S_2^* & \Rightarrow R(t_2) \geq S(t_3) & \text{(iii)} \\ & \Rightarrow R(t_2) \geq S(t_2) & \text{(ii)} \\ & \Rightarrow R(t_2) = S(t_2) & \text{(due to } R(t) \leq S(t)\text{)} \end{array}$$

Therefore, the termination has been reached at t_2. Although any processor can be the initiator, our implementation chooses Processor 0 to do that job. When seeing no

open node, it will launch a termination detection. Other processors passively reply and wait for command messages from it. The outer loop is quit if the termination condition is reached. To avoid deadlock, nonblocking communication is also used in the termination detection.

5.3 New Load Balancing

The balancer in the old library is quite simple and called through the function `ABA_OPENSUB::select()` three times before going to a normal selection. The old balancer actively sends a request for an open node from other processors. If there is a suitable node, the processor will be paused in order to wait for it. This possibly contributes much overhead to the total performance. Furthermore, the old design of the balancer is supposed to sticks firmly to the ABACUS kernel, reducing the flexibility in designing new load balancing strategies. The balancing layer should be quite independent from the computing kernel, especially in exploiting large search trees in combinatorial optimization.

A new load balancer will run independently with the kernel by being called in the branch-and-bound loop (line 21 of Algorithm 5.2.3). It can be viewed as a virtual thread which sends or receives nodes through the network and gets or puts them in or out the list of open nodes. This approach hence helps developers to easily add new load balancing methods into the parallel library. In the new parallel ABACUS, the function `select()` is similar to that of the sequential version and is only used by the ABACUS kernel. A different function named `ABA_OPENSUB::select4Remote()` is dedicated to the load balancer. Although it uses the same enumeration strategy as `select()` does, it will not return a node if the pool has only very few open nodes. It could be that the time for a node reaching a remote processor is longer than the time to solve it locally. That point makes me consider the idea not to choose the first open node in pool. Instead, a node whose position corresponds to the proportion between "time to send" and "time to solve" will be picked out. The proportion is measured within the branch-and-bound loop.

After sending an open node to another, the state of a processor changes, especially in terms of bounds, and the number of open nodes. However, if the processor considers all those factors for each remote processor immediately, this consumes much time because several state updates could be performed in one round of sharing. Moreover, some of such global state updates are unnecessary. Therefore, the new library chooses a different choice to ask a load balancing strategy to specify which node will be sent to each remote processor beforehand. Then, the parallel ABACUS does the actual sharing. If the previous sending to a remote processor has not been finished yet, the corresponding node is put back to the pool. By doing so, ABACUS developers are free to implement new load balancing methods. They only have to write new code to specify which processors will receive nodes under a new balancing strategy. The parallel ABACUS will perform the actual sending. Note, that some methods require additional

global information. In such a case, the developers should programme more functions which exchange the necessary information of the method using the mechanism of the two nonblocking functions mentioned in Section 5.2.

There are two load balancing strategies implemented in the new library:

- `Idle`: if a processor sees another idle, it will send a node to that processor immediately. Note that the sender will not send any if there is only one left in its list.

- `BestFirst`: this method also sends a node to idle processors right away. Furthermore, if not, it will check the dual bound of remote processors. If the remote bounds are much better (e.g., higher in an optimization problem) in comparison with the local bound, the local processor will send suitable nodes to the remote processors. Denote by $\underline{z}_{local}$, $\underline{z}_{remote}$, and $\underline{z}$ the dual bound of the local processor, the remote processor and the global best dual bound respectively. The tolerance Δz is given by a ABACUS parameter file under the variable `ParallelBestFirstTolerance`. If $|\underline{z}_{local} - \underline{z}| < |\underline{z}_{remote} - \underline{z}| \times \Delta\underline{z}$, a new node with the dual bound better than $\underline{z}_{remote}$ will be sent to the remote processor. The smaller $\Delta\underline{z}$, the less nodes will be transferred between processors.

In the beginning, only Processor 0 is busy to solve the root node, and all others are idle. One idea comes to force an end to the separation phase in order to generate child nodes for other idle processors. This is quite meaningful to reduce the idle time. We call it the *early branching* technique. The new parallel ABACUS supplies a parameter `nIterationsBeforeIdleCheck` to specify the number of separation iterations before checking whether to perform the early branching or not. The default value is -1 (i.e., the separation will work as the sequential ABACUS does). The condition to stop separation is when the global number of open nodes is smaller than the number of idle processors. It is reasonable not to branch early if other processors can supply enough problems for idle processors. Note that this approach can lead to a larger branch-and-bound tree. This will be shown in computational results.

5.4 Parallel Set Partitioning Solver

5.4.1 Parallelizing the sequential code

Issues concerning the sequential set partitioning solver were described in Chapter 3. Very specific points will not be presented in implementation which can distract the attention of readers. Moreover the process of parallelizing a sequential code is quite simple. All constraint and variable classes of the sequential version are added with the required pack and unpack functions. They are also implemented within new branching rule classes for the set partitioning problem. The derived class of `ABA_MASTER` has three member functions used to recreate constraints, variables (`unpackConVar()`),

nodes (`unpackSub()`), and branching rules (`unpackBranchrule()`). Following the instructions in this thesis and the report of Böhm (1999), we transform a sequential code to a parallel version with little effort.

Using a variable branching method of ABACUS, we have only two child nodes. If many processors are exploited for computation, most of them are idle waiting for few ones finishing certain separation steps or at least the first linear relaxation. In order to reduce the idle time, therefore, we apply a so-called *multi-branching* method. The number of nodes to be generated is decided based on the number of idle processors. For example, assume we have two zero-one variables x_1^f, x_2^f whose relaxation values are not integers. We can then generate the four following branching rules:

$$\begin{aligned} x_1^f = 0 \quad &\text{and} \quad x_2^f = 0, \\ x_1^f = 0 \quad &\text{and} \quad x_2^f = 1, \\ x_1^f = 1 \quad &\text{and} \quad x_2^f = 0, \\ x_1^f = 1 \quad &\text{and} \quad x_2^f = 1. \end{aligned}$$

With k variables, we can create 2^k branching rules. We also use the default ABACUS criteria to choose good branching variables. In addition, chosen variables will be sorted with respect to those criteria in order to get a needed number of best variables. The multi-branching technique is only activated if the global number of open nodes is small (similar to the idea of the early branching). Using many processors possibly prevents statistical information to be up-to-date. This possibly leads to wrong branching decisions. In such a case, many processors will simultaneously perform the multi-branching, generating more than the needed number of open nodes. Therefore, if we need m problems, and p processors have problems to be branched, the multi-branching on each processor only generates k child problems with $k < \log_2(m/n)$. With this branching method, we hope to utilize idle processors as much as possible. Remember that it is not guaranteed that a better performance will be obtained. The tree search direction will be changed much.

Finally, after being compiled and linked with the parallel ABACUS library on a parallel computer supporting MPI, a parallel executable is ready to perform computations. It is an advantage of the parallel ABACUS. The expense of designing a parallel code from a sequential one is quite small. Furthermore, the resulting code can be used on many parallel computers. The computational results of the parallel set partitioning solver will be viewed in the next section.

5.4.2 Computational results

In order to test the new MPI-based parallel ABACUS, we use the sequential branch-and-cut code discussed in Chapter 3. Readers can see more details of the sequential implementation in that chapter. We reuse the default settings for the sequential runs as following:

- Linear relaxation: CLP solver,

- Cutting plane generation: cliques and odd cycle inequalities,
- Branching: Padberg-Rinaldi with $\alpha = 0.25$; strong branching whose the number of candidates is 10,
- Node selection strategy: best first search,
- Preprocessing: all methods supported by the sequential runs,
- Heuristics: dive-and-fix and near-int-fix.

We can test with many variants of algorithms and parameters, but, in this chapter, we focus on the performance of the parallel solver which employs the new parallel ABACUS. A good combination of algorithms and parameters for the sequential tests will be used in this chapter. The parallel system for our computation is a cluster of 26 dual processor computers which are connected by a Myrinet network 1.2 Gbps. The processor type is Intel Pentium III 800 MHz. Each computer has 512 Mbyte RAM. The cluster now is integrated into HELICS which is a high performance PC cluster at the Interdisciplinary Center for Scientific Computing (IWR) of the University of Heidelberg.

Test problems have been described in Section 3.3.6. They are difficult test problems generated from crew pairing problems. As designed, the parallel ABACUS library is only helpful if the branching step is needed. Moreover, it could be a good choice for problems which have a large number of branch-and-bound nodes in sequential runs. Since different runs often give different performance results, several runs will be performed for each test problem. One run will be chosen so that it has computational statistics near the average of those of all runs.

Firstly, the performance aspects of interest should be mentioned. Tables in this report have the following format: The name of test problems is in Column 1 which is followed by the column "proc" showing the number of processors involved in computation. Columns "B&B" and "B&B exch." show the total number of branch-and-bound nodes and the number of nodes exchanged among processors, respectively. Due to anomalies in parallel search, these numbers can change significantly with different numbers of processors involved. The next "Lp" is the column of number of linear relaxations (also meaning the number of calls to cutting generation methods). The following five columns present changes of the constraint matrix in both constraint and variable dimensions. Note that columns "$m\_$" and "$n\_$" do not take into account constraints and variables removed in the preprocessing of root nodes. The next column "heu" presents how many times the heuristics find a better feasible solution. Certainly, in any parallel design, the efficiency is the most important factor to be considered. The remaining columns are dedicated to the execution time for main tasks of the parallel branch-and-bound solver. They are "$\%_{\mathrm{Pre}}$" the set partitioning preprocessing time, "$\%_{\mathrm{Sep}}$" the separation time, "$\%_{\mathrm{Heu}}$" the primal heuristics time, "$\%_{\mathrm{Bra}}$" the branching time, "$\%_{\mathrm{Lp}}$" the linear relaxation time, "$\%_{\mathrm{Par}}$" the parallel library time, "$\%_{\mathrm{Idle}}$" the idle time, and "$\%_{\mathrm{Cpu}}$" the CPU time of parallel computation. There should be a remark here on how

to collect these times. The measurement is only started after an integer formulation has been read into memory. Another comment is that all those times will not be shown in any time unit. Alternatively, they are viewed as a percentage of the total computation time which is shown in the column "t_{total}". Due to this way of presentation, it will be clearer to see how the parallel solver acts in a global view. The overhead of the library is quite interesting to observe, especially with the help of columns "$\%_{\text{Par}}$" and "$\%_{\text{Idle}}$". It is quite obvious that when the idle time increases, the number of calls to communication functions also increases. This leads to a high CPU time within the parallel library. However, the overall computation time will not be influenced. The reason for it is straightforward because the communication time increases significantly when there is no open node for computation. The last two columns help readers to get a feeling on how much communication is active among processors.

In the default settings for parallel execution, `nIterationsBeforeIdleCheck`=-1, the multi-branching is switched off, and $\Delta\underline{z} = 5.0\%$. With these default settings, we obtain the computational results in Tables 5.1 and 5.2. A quite good performance is obtained. The most important target of the design has been achieved concerning the improvement of the computation time in most cases. With a sufficient number of processors, the total computation time is reduced for all test problems, except "us01". If the idle percentage is not so high, the speedup is quite good. However, the efficiency of the parallel solver is strongly negatively affected if there are few branch-and-bound nodes, such as in "us01", "vncpp1", "vncpp2", and "vncpp3". Although there are time reductions, the speedup of the parallelization is not high. Problem "us01" also has an additional reason. Since this problem possesses a very large matrix, it is easy to understand that we cannot easily obtain a faster computation with this way of parallelization in which the smallest processing unit is a branch-and-bound node (see more in Section 5.1.2). With problems which have a very large constraint matrix and need few branch-and-bound nodes, the matrix decomposition is a better parallelization approach.

In order to consider the performance of the parallel ABACUS, a branch-and-bound setting is used. The primal heuristics are also excluded to guarantee as little effects of randomness as possible during long computations. The number of candidates for the strong branching is reduced to 2. With those settings, sequential runs take quite a long time to obtain an optimal solution. A speedup for each test problem is visualized in Figure 5.2. Note, that the problem "us01" is not our concern due to the small number of nodes. Then, we receive quite good speedups for time-consuming computations, such as "aa01", "aa04". The super-linear speedup in case of "aa04" is not within the expectation, but it happens sometimes. The thesis wants to emphasize an interesting point concerning the idle percentage visualized in the right picture. If there are still enough open nodes to supply for idle processors, we still have a performance improvement ("aa04"). The parallel version of ABACUS library demonstrates its efficiency through the set partitioning solver.

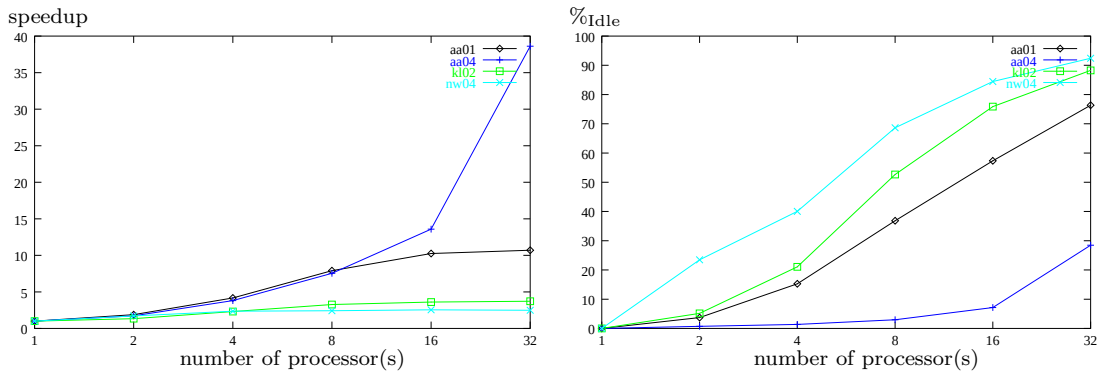
Now, we will try to investigate in more details why the poor performance happens when using many processors. Since the parallel design is based on the idea of polling

Name	proc	B&B		Lp	clique	odd	m^+	n^-	m^-	heu	time									traffic	
		total	exch.			cycle					$\%_{\text{Pre}}$	$\%_{\text{Sep}}$	$\%_{\text{Heu}}$	$\%_{\text{Bra}}$	$\%_{\text{Lp}}$	$\%_{\text{Par}}$	$\%_{\text{Idle}}$	$\%_{\text{Cpu}}$	t_{total}	#msg	MB
aa01	1	143	0	686	1239	640	549	603	413	3	8.43	1.20	18.62	1.58	69.98	0.00	0.00	99.59	0:17:32	0	0.00
aa01	2	192	29	738	1790	867	587	676	497	3	7.74	1.20	16.23	1.40	65.86	3.03	7.22	97.98	0:14:30	360	9.47
aa01	4	199	52	764	1877	926	624	361	265	1	5.39	1.36	7.35	1.15	52.69	18.11	31.47	95.29	0:05:57	956	16.99
aa01	8	249	90	780	2102	1046	623	300	247	1	3.15	0.84	5.36	0.69	32.03	39.89	57.32	93.99	0:05:30	2762	29.58
aa01	16	214	75	694	2855	1445	557	294	236	1	1.52	0.51	2.72	0.32	15.00	63.35	79.74	95.08	0:05:23	6410	25.47
aa01	32	217	84	843	4167	2238	713	337	319	1	0.78	0.42	1.35	0.16	7.61	77.02	89.48	96.73	0:05:16	13440	32.65
aa04	1	303	0	1180	1480	839	955	844	408	3	10.46	2.04	13.76	2.06	71.18	0.00	0.00	99.55	0:17:15	0	0.00
aa04	2	374	61	1087	1452	931	902	1078	476	5	10.86	1.93	16.40	2.11	64.04	2.08	4.27	98.54	0:12:09	583	15.99
aa04	4	508	93	1578	1963	1189	1290	1037	555	4	12.24	1.24	16.88	1.92	59.27	5.01	8.36	98.28	0:11:08	1936	24.50
aa04	8	509	150	1517	3526	2162	1263	690	454	7	7.40	2.19	12.76	1.47	45.27	21.34	30.21	96.65	0:04:33	4019	39.70
aa04	16	707	318	1951	4813	3223	1596	740	546	6	4.45	1.43	6.78	0.78	26.70	47.76	59.14	96.21	0:04:12	13422	86.28
aa04	32	634	259	1846	6584	4212	1503	781	562	6	2.43	0.90	3.69	0.42	13.85	68.30	78.36	97.12	0:03:53	28874	79.56
kl02	1	21	0	73	326	248	62	0	0	2	43.05	1.38	5.64	1.59	47.77	0.00	0.00	99.60	0:03:37	0	0.00
kl02	2	17	4	54	266	220	43	466	8	0	38.76	1.11	1.51	0.97	31.21	9.56	25.17	95.01	0:01:47	38	2.79
kl02	4	25	8	71	316	269	58	0	0	0	25.45	0.73	0.75	0.68	21.18	26.99	50.28	93.44	0:01:47	165	5.58
kl02	8	25	8	71	316	269	58	0	0	0	12.59	0.37	0.37	0.34	10.51	48.82	75.12	92.66	0:01:47	423	5.59
kl02	16	25	8	71	316	269	58	0	0	0	6.05	0.17	0.18	0.16	5.01	62.85	87.89	95.12	0:01:53	911	5.62
kl02	32	33	12	67	235	257	50	20	10	0	2.73	0.06	0.10	0.08	2.43	73.53	94.29	96.92	0:02:05	2551	8.61
nw04	1	177	0	900	4203	96	723	68	11	4	21.77	38.02	4.90	0.41	26.45	0.00	0.00	99.92	0:20:22	0	0.00
nw04	2	148	31	623	2868	67	506	262	19	3	9.95	28.35	3.82	0.46	26.07	11.85	23.18	94.86	0:09:46	351	62.70
nw04	4	165	58	611	3059	74	506	819	9	2	8.76	15.44	2.17	0.30	15.13	33.04	53.35	92.11	0:08:37	1304	118.72
nw04	8	188	65	819	4697	144	696	70	7	2	3.95	10.06	1.15	0.15	8.03	53.00	73.73	92.43	0:08:22	3612	137.35
nw04	16	326	113	1443	8370	196	1232	0	4	2	2.39	7.62	0.65	0.07	4.14	66.58	83.03	94.78	0:10:42	12589	250.81
nw04	32	243	86	1190	7589	222	1034	0	4	2	1.18	2.36	0.25	0.04	1.79	80.46	93.56	96.68	0:10:42	19575	193.59
us01	1	9	0	65	516	157	60	0	0	0	12.84	5.04	6.52	0.98	64.60	0.00	0.00	91.14	1:06:41	0	0.00
us01	2	20	5	105	663	221	96	0	0	0	9.44	3.38	3.04	0.64	40.73	16.47	36.46	88.41	1:08:47	52	75.15
us01	4	19	6	121	822	269	111	0	0	0	5.54	2.14	1.63	0.33	21.49	33.37	65.35	86.82	1:03:19	156	91.95
us01	8	19	6	118	817	257	108	0	0	0	2.65	0.99	0.79	0.16	10.24	51.20	82.86	88.26	1:06:05	380	91.97
us01	16	13	4	79	543	155	72	0	0	0	1.07	0.34	0.42	0.05	4.30	65.40	92.43	91.56	1:03:35	601	61.26
us01	25	10	3	71	505	155	65	0	0	0	0.62	0.20	0.26	0.03	2.57	70.18	95.10	93.61	1:03:42	763	46.07

Table 5.1: Computational results of the parallel set partitioning solver with the default settings

Name	proc	B&B		Lp	clique	odd	m^+	n^-	m^-	heu	time									traffic	
		total	exch.			cycle					%Pre	%Sep	%Heu	%Bra	%Lp	%Par	%Idle	%Cpu	t_{total}	#msg	MB
vncpp1	1	19	0	44	209	179	35	0	0	2	37.28	1.36	3.96	1.38	54.30	0.00	0.00	99.77	0:10:35	0	0.00
vncpp1	2	35	6	76	496	321	60	173	76	2	19.92	1.35	8.60	1.15	43.32	10.60	23.18	95.23	0:11:01	50	23.63
vncpp1	4	57	20	98	451	418	77	156	94	6	18.69	0.91	2.97	0.86	31.89	24.44	41.78	94.23	0:09:13	327	78.76
vncpp1	8	69	26	113	564	488	89	222	137	8	9.18	0.50	2.22	0.47	17.39	46.93	68.46	93.37	0:09:21	956	102.42
vncpp1	16	61	22	95	475	403	74	0	0	7	5.77	0.24	0.96	0.25	8.21	60.67	83.46	95.24	0:08:39	1902	86.74
vncpp1	32	55	20	96	450	387	75	159	96	5	2.33	0.10	0.33	0.10	3.84	72.57	92.69	96.95	0:09:16	3466	79.19
vncpp2	1	3	0	20	118	84	17	5	39	1	26.37	1.07	24.85	0.33	45.76	0.00	0.00	99.79	0:05:51	0	0.00
vncpp2	2	4	1	22	133	96	19	0	0	1	16.40	0.60	12.41	0.16	23.21	17.16	45.96	91.62	0:05:58	16	3.83
vncpp2	4	4	1	22	133	96	19	0	0	1	8.06	0.30	6.25	0.08	11.44	37.07	73.05	90.74	0:05:59	46	3.83
vncpp2	8	4	1	22	133	96	19	0	0	1	3.93	0.15	3.03	0.04	5.76	53.12	86.55	92.56	0:06:07	106	3.83
vncpp2	16	4	1	22	133	96	19	0	0	1	1.84	0.07	1.45	0.02	2.68	61.09	93.31	95.20	0:06:33	212	3.84
vncpp2	32	4	1	22	133	96	19	0	0	1	0.89	0.03	0.68	0.01	1.32	66.99	96.66	97.19	0:06:48	406	3.87
vncpp3	1	13	0	39	206	165	30	392	60	1	37.52	1.34	3.73	0.98	54.41	0.00	0.00	99.74	0:07:45	0	0.00
vncpp3	2	16	3	35	174	135	24	163	67	1	24.48	0.74	2.23	0.58	34.23	15.68	35.87	92.45	0:06:27	37	10.09
vncpp3	4	19	6	38	202	153	27	94	152	1	12.51	0.48	1.29	0.35	20.58	35.20	63.09	90.82	0:05:29	166	20.19
vncpp3	8	19	6	38	202	153	27	94	152	1	6.11	0.23	0.63	0.17	10.11	54.75	81.72	91.98	0:05:35	388	20.20
vncpp3	16	19	6	38	202	153	27	94	152	1	2.87	0.11	0.31	0.08	4.76	64.92	91.02	94.70	0:05:55	826	20.21
vncpp3	32	19	6	38	202	153	27	94	152	1	1.40	0.05	0.15	0.04	2.33	72.17	95.54	96.91	0:06:06	1582	20.28
vncpp4	1	93	0	330	782	448	244	1134	286	2	5.72	2.46	5.65	2.44	80.17	0.00	0.00	99.56	0:22:31	0	0.00
vncpp4	2	45	6	113	517	268	74	571	256	1	9.10	1.61	3.73	1.63	59.79	9.24	20.74	95.73	0:08:00	74	18.31
vncpp4	4	77	26	148	510	315	97	554	248	1	4.73	0.88	2.32	0.98	34.10	31.64	53.71	92.11	0:07:46	571	79.27
vncpp4	8	155	54	336	1100	634	235	652	298	1	2.13	0.69	1.26	0.71	27.58	45.61	64.88	93.55	0:09:39	2307	164.67
vncpp4	16	145	50	298	1125	659	203	602	306	1	0.93	0.29	0.74	0.29	10.39	66.10	86.09	94.74	0:10:36	5435	152.94
vncpp4	32	80	27	184	970	526	131	558	282	1	0.56	0.15	0.32	0.11	4.14	75.41	94.12	96.90	0:08:14	6413	83.28

Table 5.2: Computational results for randomly generated test problems



(a) Speedups

(b) $\%_{\text{Idle}}$

Figure 5.2: The speedups and $\%_{\text{Idle}}$ of the parallel set partitioning solver in branch-and-bound mode

through MPI communication objects to check their completions, the overhead of the parallel ABACUS library naturally increases if there are very few open nodes globally which are not satisfied by many idle processors. This often occurs in the beginning or at the end of a run. Moreover, in a branch-and-bound search, there are possibly points of time when only very few open nodes exist. This also reduces much the utilization of all processors. Columns "$\%_{\text{Par}}$" and "$\%_{\text{Idle}}$" show the increase of the parallel overhead and the idle time corresponding to the increase of the number of processors. From the table, we imply that most of the parallel overhead is due to the idleness of processors. The table also shows that if we use more processors, the idleness increases and it has a strong impact on the speedup. With such problems, one way to improve performance even further is to use a different approach of parallelization. Being different from a simple branch-and-bound (without cutting plane generation), the time for each node in the set partitioning solver is longer. Therefore, the idleness is also larger as a result. The tree search for the problem "nw04" is visualized in Figure 5.3.a as an example. Without any load balancing mechanism, we easily see in the figure that the processors often wait for receiving an open node, especially at the end. The figure depicts that processor 0 has no jobs for a long time. Furthermore, the search tree is narrow in width. Although all the processors want to join into the computation, there is no problem to supply them with. As mentioned before, the speedup cannot be improved if we use more processors. The situation also occurs with the randomly generated test problems.

However, with the use of nonblocking communication, the library can spend much of computing power for solving problems. The column "$\%_{\text{Cpu}}$" indicates that a quite small part of the total time is used for operating system. Moreover, the overhead of the parallel library is mainly due to using too many processors inappropriately. An interesting point concerns the traffic exchanged among processors. We see that the maximum volume of exchanged data is about 250MB. Certainly, with a high speed

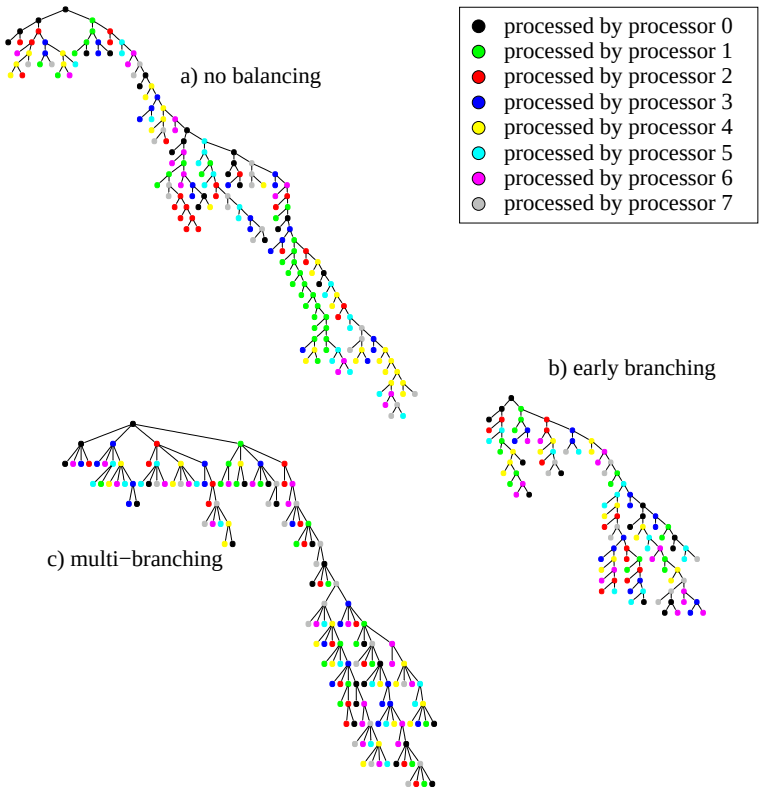


Figure 5.3: Search trees of "nw04" using 8 processors

network like Myrinet, transferring such volume takes a small time (about 2 seconds in a global view). If we use a network of 100Mbps which is quite popular nowadays for cluster systems, the needed time is also small in comparison with the total computation time. This proves the efficiency of the parallel design.

In other computations, we will see the effect of the load balancing techniques mentioned before. The first technique is the early branching which forces a processor to stop its separation immediately when realizing there are not enough nodes for idle processors. As seen in the computation using the default settings, processors are often idle because there are not enough problems for them. It is expected that the early branching technique will do a better job in the same situation. The parameter `nIterationBeforeIdleCheck` is used to control the length of idleness. We should be careful with this parameter. Stopping the cutting plane generation very early is possibly not a good choice because doing that makes a branch-and-cut code only execute a simple branch-and-bound algorithm. But we can receive a benefit by reducing the idle time of processors. Computational results in Table 5.3 are obtained with the default settings, except that `nIterationBeforeIdleCheck`=0. There are 2 interesting columns: "B&B" and "$\%_{\text{Idle}}$". There are usually more numbers of branch-and-bound nodes in comparison with the default computation before. This is within the expectation because the branching will be performed earlier. Another different point relates

to the reduction of the idle time. These two factors often have a tradeoff relationship and both affect the total computation time. But the total computation time is smaller in most cases. Figure 5.3.b shows a search tree of "nw04" which is quite different to Figure 5.3.b . Processors are involved more often in any level of the tree. The early branching works effectively with the test problems. There is quite an interesting point on the cutting plane separation for "kl02", "nw04" and "us01" when using more processors than needed. Although certain clique and odd cycle violated inequalities have been generated, they are not so much necessary for the solver to find the optimal solution.

Another balancing method in examination is the multi-branching method which will generate enough nodes for idle processors. The third tree in Figure 5.3 shows all processors are to be utilized at any stage of the computation. Even though there are very few open nodes at some time, the multi-branching method also prefers to creating jobs for idle processors. Remember that, if we branch on many variables at a time, the feasible region of the child nodes is stricter in comparison with branching on one variable only. The reason is quite obvious as we fix many variables at a time. Then, the multi-branching helps to reduce the tree depth a lot (e.g., the depth of the third tree is half of that of the first tree for "nw04"). The number of branch-and-bound nodes is usually increased this much according to the number of processors, as shown by Column "B&B" in Table 5.4. Not as the default settings, the multi-branching method makes processor more busy (Column "$\%_{\text{Idle}}$"). Although there are more branch-and-bound nodes, the time for each node is likely to be smaller because the feasible region of a problem is smaller and the separation for violated global inequalities often stops earlier. Unfortunately, the multi-branching is not very helpful in the experiments. We see that t_{total} is smaller for some tests, but larger in others. Logically, the multi-branching would be only effective in the case of narrow branch-and-bound trees. This just happens sometimes in the test set (e.g., "nw04"). Moreover, solving an "easy" problem by linear relaxation and cutting plane separation is possibly faster than branching it and sending its children to other processors for solving. Working with problems of a large-sized matrix, the multi-branching can also make computers run out of memory very quickly. Therefore, one should be careful when using this technique.

The performance in Tables 5.3 and 5.4 demonstrates that the design expectations for most of the test problems have been fulfilled partly, especially the early branching. The smaller idle time of processors is followed by a better performance. Although the anomalies in a parallel search is inherent to tree search, two recommended load balancing techniques are little effective for the parallel set partitioning solver. They need more testings, not only for other classes of problems, but also for set partitioning problems (possibly not from the crew pairing problem).

Along with the change of the communication library from ACE to MPI, the new parallel ABACUS has proven its effectiveness in parallelizing a sequential code. With little knowledge on parallel programming, users can perform a big computation on parallel computers. The new design of the library employs nonblocking communication which increases the overlap between computation and communication. Each kind

Name	proc	B&B		Lp	clique	odd	m^+	n^-	m^-	heu	time									traffic	
		total	exch.			cycle					%Pre	%Sep	%Heu	%Bra	%Lp	%Par	%Idle	%Cpu	t_{total}	#msg	MB
aa01	1	143	0	686	1239	640	549	603	413	3	8.50	1.22	18.60	1.58	69.91	0.00	0.00	99.59	0:17:21	0	0.00
aa01	2	174	23	664	1365	680	529	550	317	4	6.92	1.18	15.51	1.46	67.44	3.06	7.19	98.07	0:12:44	318	7.50
aa01	4	218	59	637	1940	1002	495	295	235	2	4.84	1.19	7.99	1.12	57.19	15.54	27.14	95.98	0:06:28	955	19.25
aa01	8	356	139	946	2772	1516	755	660	465	2	4.48	0.88	7.37	0.95	41.33	30.95	44.48	95.50	0:05:46	3194	45.45
aa01	16	387	146	458	1933	928	230	465	395	3	1.98	0.27	3.81	0.47	19.08	58.66	74.13	95.40	0:05:06	7731	47.90
aa01	32	415	154	283	455	228	39	488	421	2	1.08	0.03	1.97	0.25	9.60	74.76	86.94	96.93	0:05:03	16202	50.23
aa04	1	303	0	1180	1480	839	955	844	408	3	10.54	2.03	13.88	2.04	71.02	0.00	0.00	99.55	0:17:08	0	0.00
aa04	2	387	58	1328	1769	984	1077	921	452	1	12.05	1.24	16.12	2.09	65.99	1.20	2.39	99.03	0:15:26	636	15.22
aa04	4	497	92	1518	2614	1618	1241	931	559	5	10.01	2.28	11.25	2.05	65.44	5.05	8.41	98.39	0:07:22	1842	24.22
aa04	8	508	157	1477	4455	2628	1210	848	416	5	7.56	2.17	9.97	1.60	49.88	19.99	28.18	96.90	0:04:22	4049	41.54
aa04	16	916	427	1817	5341	3345	1403	1112	623	4	4.66	1.62	5.63	1.03	34.19	41.44	51.60	96.73	0:03:16	13141	114.54
aa04	32	696	277	934	4226	2341	545	1115	566	1	2.57	0.54	2.80	0.52	16.03	66.24	77.13	97.19	0:02:48	23942	75.81
kl02	1	21	0	73	326	248	62	0	0	2	43.39	1.38	5.60	1.57	47.56	0.01	0.00	99.67	0:03:36	0	0.00
kl02	2	18	5	26	76	89	15	910	18	1	32.53	0.62	2.16	0.75	30.24	11.43	31.39	94.79	0:01:01	38	3.46
kl02	4	25	8	71	316	269	58	0	0	0	25.41	0.73	0.77	0.69	21.25	26.63	50.26	93.41	0:01:47	165	5.58
kl02	8	43	14	24	0	0	0	1237	22	1	8.57	0.00	0.68	0.22	8.21	49.79	81.04	92.89	0:01:06	584	9.63
kl02	16	40	13	23	0	0	0	1127	21	1	3.91	0.00	0.31	0.10	3.81	59.57	90.93	95.40	0:01:10	1224	8.95
kl02	32	43	14	24	0	0	0	1237	22	1	1.89	0.00	0.14	0.05	1.73	66.04	95.42	96.98	0:01:13	2074	9.64
nw04	1	177	0	900	4203	96	723	68	11	4	21.85	38.04	4.91	0.41	26.32	0.00	0.00	99.91	0:20:23	0	0.00
nw04	2	200	35	586	2778	120	421	23	8	2	17.64	20.36	4.33	0.34	24.48	12.33	22.82	94.90	0:09:29	413	69.72
nw04	4	317	110	381	1413	80	175	81	18	3	15.64	5.93	3.64	0.30	19.70	30.87	44.36	93.76	0:06:04	1887	216.48
nw04	8	174	61	122	79	8	8	73	10	4	9.82	0.16	1.56	0.20	10.06	52.86	74.47	92.71	0:04:06	2408	118.90
nw04	16	231	80	152	0	0	0	70	6	5	5.47	0.00	0.84	0.11	5.58	64.13	85.30	95.08	0:04:03	6554	155.18
nw04	32	251	92	159	0	0	0	0	4	6	2.53	0.00	0.39	0.05	2.51	74.91	93.11	96.92	0:04:35	12663	178.51
us01	1	9	0	65	516	157	60	0	0	0	13.03	5.15	6.55	0.98	65.46	0.00	0.00	92.35	1:05:12	0	0.00
us01	2	12	3	48	334	73	40	0	0	0	9.17	1.48	9.09	0.31	31.49	18.46	43.16	87.66	0:58:58	34	45.15
us01	4	16	5	19	87	21	9	0	0	0	5.96	0.19	4.48	0.18	16.15	35.21	69.69	86.18	0:58:48	118	73.57
us01	8	25	8	13	0	0	0	0	0	0	2.75	0.00	2.12	0.13	8.53	51.66	84.21	88.08	1:03:28	394	117.71
us01	16	22	7	13	0	0	0	0	0	0	1.47	0.00	1.07	0.06	4.18	64.81	91.77	91.65	1:01:34	805	103.00
us01	25	25	8	14	0	0	0	0	0	0	0.85	0.00	0.57	0.04	2.40	71.30	94.45	93.19	1:13:27	1311	117.72

Table 5.3: Computational results of the parallel set partitioning solver with early branching

Name	proc	B&B		Lp	clique	odd	m^+	n^-	m^-	heu	time									traffic	
		total	exch.			cycle					%Pre	%Sep	%Heu	%Bra	%Lp	%Par	%Idle	%Cpu	t_{total}	#msg	MB
aa01	1	143	0	686	1239	640	549	603	413	3	8.43	1.21	18.62	1.58	69.97	0.00	0.00	99.60	0:17:32	0	0.00
aa01	2	193	34	702	1743	839	557	634	512	3	7.88	1.13	16.48	1.46	64.85	3.43	7.94	97.85	0:14:07	353	11.10
aa01	4	212	55	799	2037	1002	648	363	273	1	5.39	1.41	7.37	1.15	52.71	18.07	31.30	95.09	0:06:08	984	17.98
aa01	8	299	136	819	2705	1406	687	415	319	2	3.76	1.15	4.19	0.70	38.88	35.39	50.56	94.81	0:04:54	2747	44.67
aa01	16	540	259	1141	2383	1242	871	1321	1116	1	4.16	0.57	1.42	0.52	31.54	47.72	60.99	96.24	0:04:14	10561	86.21
aa01	32	1557	932	2984	4214	1861	2435	2151	3241	2	4.53	0.53	0.76	0.52	33.23	50.29	59.21	97.85	0:04:34	34091	324.54
aa04	1	303	0	1180	1480	839	955	844	408	3	10.45	2.01	13.89	2.05	71.17	0.00	0.00	99.58	0:17:16	0	0.00
aa04	2	303	46	965	1352	843	772	998	506	4	11.59	1.81	17.22	2.00	62.70	2.08	4.43	98.49	0:11:47	481	12.08
aa04	4	518	103	1691	2134	1306	1421	675	673	6	12.98	1.34	16.57	1.97	58.84	5.05	8.31	98.41	0:10:55	1989	27.14
aa04	8	854	207	2912	4257	2534	2354	1698	1191	7	9.73	1.94	11.09	1.78	58.70	11.46	15.97	97.99	0:06:42	6378	54.81
aa04	16	1905	978	4788	5914	3671	4022	1817	1243	3	8.38	2.39	4.40	1.51	53.14	23.37	28.10	97.88	0:04:47	22832	260.75
aa04	32	4461	3008	8151	9755	5798	7007	3723	3356	4	10.33	2.02	3.37	1.36	50.13	27.78	30.46	98.63	0:05:07	73300	839.25
kl02	1	21	0	73	326	248	62	0	0	2	42.99	1.36	5.54	1.59	47.83	0.00	0.00	99.48	0:03:37	0	0.00
kl02	2	17	4	54	266	220	43	466	8	0	38.28	1.10	1.51	0.99	30.70	9.99	25.13	93.94	0:01:47	38	2.79
kl02	4	25	8	71	316	269	58	0	0	0	25.34	0.73	0.76	0.68	21.26	26.87	50.30	92.91	0:01:47	167	5.58
kl02	8	101	42	196	779	655	140	1224	62	0	19.36	0.89	0.76	0.36	14.22	40.34	62.48	94.23	0:01:43	1500	29.16
kl02	16	254	121	305	858	926	193	2518	180	0	21.10	0.49	0.24	0.35	12.06	45.03	63.76	96.44	0:02:07	5109	84.63
kl02	32	426	191	484	1073	1366	283	6656	302	0	8.03	0.54	0.25	0.13	6.62	57.01	81.91	97.59	0:01:21	16714	134.73
nw04	1	177	0	900	4203	96	723	68	11	4	21.82	38.08	4.89	0.41	26.29	0.00	0.00	99.89	0:20:19	0	0.00
nw04	2	138	29	713	3547	91	604	312	23	3	12.45	28.48	3.44	0.42	22.54	12.72	25.59	94.23	0:11:34	349	58.81
nw04	4	193	70	755	3766	149	633	813	7	2	8.42	19.23	2.06	0.27	14.23	31.78	50.44	92.41	0:09:48	1453	142.33
nw04	8	293	118	864	4544	173	698	49	20	2	5.35	12.51	1.06	0.21	11.72	47.68	64.51	93.46	0:07:29	4333	247.39
nw04	16	392	177	630	3105	83	435	0	28	2	3.85	3.87	0.53	0.07	6.16	65.70	82.52	94.97	0:06:42	11032	375.81
nw04	32	832	389	1015	4765	92	646	0	64	2	2.48	2.95	0.27	0.03	3.47	75.13	88.20	96.93	0:07:14	41792	823.30
us01	1	9	0	65	516	157	60	0	0	0	13.09	5.18	6.55	1.00	65.89	0.00	0.00	92.89	1:04:46	0	0.00
us01	2	20	5	105	663	221	96	0	0	0	9.26	3.28	2.98	0.62	39.65	16.67	36.70	87.09	1:10:25	52	75.15
us01	4	19	6	120	820	263	110	0	0	0	5.23	2.02	1.55	0.32	20.49	33.86	65.58	85.26	1:06:13	155	91.95
us01	8	15	6	50	408	57	41	0	0	0	2.04	0.57	1.00	0.04	8.25	52.69	85.95	88.06	0:52:38	265	94.23
us01	16	22	9	74	514	150	61	0	0	0	1.41	0.34	0.46	0.04	4.39	65.14	90.93	91.10	0:58:09	800	140.67
us01	25	66	31	87	545	120	61	0	0	0	0.75	0.07	0.10	0.01	1.12	77.80	92.98	89.91	2:51:48	2899	492.78

Table 5.4: Computational results of the parallel set partitioning solver with multi-branching

of data exchanged between processors has been considered appropriately in order to reduce the library overhead. The MPI design also facilitates extensions to include more complicated communication data. Certainly, aspects of performance cannot be forgotten. We observed the performance of the new library by parallelizing quickly a sequential set partitioning solver. With a suitable number of processors, solving a set of well-known test problems in parallel, the library brings down significantly the running time of all hard problems with the default settings. Good speedups could be obtained with problems having a large branch-and-bound tree. There are two more load balancing techniques added, one to the parallel library, and another to the application code. The latter is possibly embedded into the ABACUS library. They help to reduce the idle time of processors, and, hence, improve the performance of the parallel runs.

There are still some limits in capability. However, the library can be enhanced and more features can be added easily. We encourage users to test our new MPI-based parallel library in order to get feedback on design and possible bugs.

Chapter 6

A Parallel Pricing for the Branch and Price Approach

When solving crew scheduling problems by column generation, the main task is to solve the pricing problem in order to introduce new columns. This problem is $\mathcal{NP}$-hard and usually requires more than 90% of the overall computation time in all of our experiments performed in Chapter 4 as well as in experiments reported in the literature. Therefore it is critical to achieve good performance in this step. Moreover, after reading Chapter 5, one can see that the parallelization on the level of branch-and-bound is only useful with problems having a large number of unsolved nodes at a time. This does not seem to happen often in case of the considered crew pairing problems. The parallel ABACUS has not supported yet the local variables/constraints. Therefore, this chapter discusses an approach of using a cluster of computers to solve the pricing problem. Several aspects of parallelizing the pricing step are investigated and computational results are reported. The parallel algorithms will be designed in such a way that they facilitate extensions and generalizations. More details can be found in Hoai et al. (2003).

6.1 Parallelizing Sequential Pricing Algorithms

Many algorithms have been suggested to solve the pricing problem to optimality. Desrosiers et al. (1995) discuss many aspects of using resource constrained shortest path algorithms. Another way is to use k shortest path algorithms to find the k most negative reduced cost variables. After ranking paths between two given nodes by k shortest paths algorithms, a rule and regulation checking procedure will eliminate infeasible pairings. An application of a k shortest paths algorithm is developed by the CARMEN systems to solve the crew scheduling problem (see Gustafsson, 1999). Constraint logic programming prevails with the ability to model these rules quickly and easily. General approaches are discussed in Guerinik and Caneghem (1995) and there are many papers concerning the use of CLP in crew management operations. In Chapter 4, you can find more information about working on the pricing step of the crew pairing problem.

There have already been efforts to use parallel computers for solving crew scheduling problems. In PAROS (Alefragis et al., 1998), the parallelization is not implemented in the process of solving the linear relaxation problem of the master problem. Instead, PAROS distributes the enumeration of pairings over the processors and their outputs will be fed into a set covering optimizer which is an iterative Lagrangian heuristic. The set covering optimizer was also parallelized, but not as successful in terms of efficiency as the enumeration phase. The parallel algorithm shows good performance when solving some real world problems of Lufthansa. Another system employing parallel computing is RALPH of Marsten (1997), but, unfortunately, there is no published report available. A further idea for using parallel computers was suggested in Klabjan and Schwan (1999). Here the huge number of feasible pairings is generated in parallel and then used to construct constraint matrices of set partitioning problems. These problems are then solved by branch-and-cut approaches. Finally, Chapters 3 and 5 of this thesis present an approach to solve large scale set partitioning problems coming from crew pairing problems by high performance facility.

Now we will revisit several sequential pricing algorithms in order to determine the ones suitable for parallelization. Methods for solving the pricing problem can be classified into the categories (as presented in Chapter 4):

- resource constrained shortest path problem,
- k shortest paths problem,
- constraint logic programming,
- hybrid approach.

The two first methods are based on graph algorithms. Although much work has been done in the area of parallel shortest paths, the theoretical worst case execution times of these parallel algorithms have the same bounds as those of the sequential ones (see Awerbach and Gallager, 1985). These algorithms only work well on sparse or regular graphs which are efficiently partitioned into subgraphs having few boundary nodes. Unfortunately, this is not the case for the flight graphs of crew pairing problems. Moreover, there has been little effort for parallelizing resource constrained shortest paths and k-shortest paths algorithms on distributed memory machines. Most of them are concerned with shared memory systems. For example, Ruppert (2000) suggests an algorithm with the theoretical concurrent-read exclusive-write PRAM model. This is mainly due to the fact that these algorithms have a strong data dependency among computing nodes which is only inefficiently implemented on distributed memory parallel computers. Another disadvantage of these methods is that they are not well suited for extensions. Since airlines rules must be embedded into graphs, including a new rule requires the reconsideration of the graph structure and algorithms as well. This difficulty also occurs in sequential algorithms.

Surprisingly, enumeration turned out to be a good choice for parallelization although it is the worst solution to many combinatorial optimization problems. Enumeration

is parallelized in a straightforward way in which the rule checking can be treated separately as a black box. The single jobs are independent, whether we use implicit (like constraint programming) or explicit (like exhaustive enumerating) approaches. They can be shared among processors of a cluster without any data dependency. The most effective parallel model for our problem is the master-slave model. In addition to the divide-and-conquer framework, we also use a bounding technique which is helpful to reduce the search region if we prefer to find the most negative reduced cost. Moreover, the thesis will present a framework which can be used to embed different categories of pricing methods, provided that the pricing subproblem is decomposed into small ones.

6.2 Aspects of Implementation

For easy scalability, in our master-slave model, we dedicate processor 0 to be only responsible for distributing jobs, waiting for results and running the branch-and-cut kernel based on the framework ABACUS. Note that the job of a slave in this approach consists of finding negative reduced cost pairings starting from a given flight. Certainly, a unit job is large enough with respect to communication.

With enough input parameters, a slave can use any sequential pricing method to solve its problem. A general algorithm for the master is designed to work with any pricing method. Choosing the pricing method on slaves freely is helpful for further extensions. If we need to find the most negative reduced cost in each iteration, we can also easily apply a bounding technique due to the centralized control. Remember that, in this case, the initial lower bound is zero because we only find negative reduced cost pairings.

In the enumeration scheme, the idea, mentioned in Section 4.3.4, of restarting the process from the flight leg right after the one where the previous iteration stopped is re-used. There is a difference in how to deal with flights belonging to the found pairings on the search process. As soon as a pairing is found, the sequential algorithms remove flights of that pairing from the search domain. The parallel version does the same in the master-slave model by sending flight identifications of found pairings to slaves in order not to consider them anymore. However, the order of found pairings will not be the same as that of the sequential algorithms.

Since the computation time of each job is nondeterministic, we can experience an unfortunate behavior of the slaves. Namely, it can happen that, although the termination condition has been reached (e.g., enough new pairings have been generated or there is no more flight leg to be sent to slaves), the master still must wait for the completion of all slaves in a column generation iteration. If a slave has been assigned a difficult task, then the master also must suspend its further activities. This does not have much impact on the performance in the beginning when it is quite easy to find enough negative reduced cost pairings. However, in the middle and final stages of column generation, much of overhead is induced by this phenomenon. Stopping early enough the waiting process of the master could be a solution. However, it should be

carefully implemented to guarantee the correctness of a used technique and enhance the performance.

A numbering technique is employed and Algorithm 6.2.1 shows its outline. Its aim is to reduce the idle times of slaves between the solution of the current linear programming relaxation and the pricing step. The main idea of the technique is to early stop long waiting periods of the master and to use sequence numbers to keep track of valid results. Each computing job sent to slaves is accompanied by a sequence number s_{master}. The slave must keep this number and will send back its number to the master. The received sequence number is stored in s_{slave}. This number is compared with the current sequence number on the master (line 18). If they match, the received result is valid for the current iteration, otherwise, we discard the result. The code segment of lines 10–15 has the consequence, that if t_{free} (i.e., the duration between the time when the termination condition was reached and the current time) is large, then the algorithm will stop waiting and go immediately to the linear programming relaxation. The length of that period is controlled by the ratio `IdleRatio`. This ratio should be kept small (e.g., 0.1%). Note that the variable s_{master} must be a global variable because its value will be used in the next iteration of the outer column generation method. Certainly, we must increase s_{master} after quitting the loop.

Figure 6.1 visualizes the states of 9 processors involved in solving the problem "vircpp1". The figure only shows a column generation iteration in which the master does not wait for the completion of several processors. While the master is computing the new LP relaxation, the slaves still perform their old pricing subproblems. Note, that the time periods which are not mentioned by any color in the legend are LP relaxations on the master, or waiting phases on the slaves.We will see later this behavior reduces the idle time of slaves.

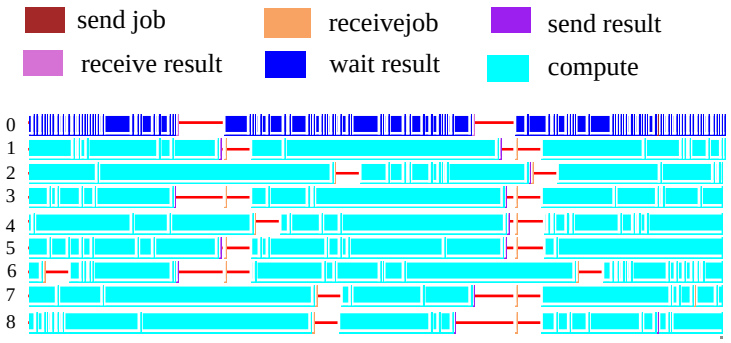


Figure 6.1: Performance visualization of the sequence numbering technique, using 9 processors to solve "vircpp1"

The algorithm tells that a received result will be removed if it does not match the current sequence number. However, in order to save them, the master possibly reuses those having a negative reduced cost in the new dual solution. This is accomplished in the implementation. Besides, parallelizing the sequential heuristic enumeration, we also have to take care of the preprocessing process (e.g., pre-optimization) on slaves in

Algorithm 6.2.1 Sequence numbering technique

1: $t_{\text{busy}} \longleftarrow \infty$.
2: Start timer.
3: **while** ($\neg$(terminate condition reached)$\vee$(some processor p busy)) **do**
4: **for all** idle processor p **do**
5: **if** ($\neg$(terminate condition reached)) **then**
6: Send s_{master} to p.
7: Send a job to p.
8: **end if**
9: **end for**
10: **if** ((some slave idle)$\wedge$(#pairing found > 0)) **then**
11: $t_{\text{free}} \longleftarrow$ current time
12: **if** ($t_{\text{free}}/t_{\text{busy}} \geq$ `IdleRatio`) **then**
13: Stop timer and quit the loop.
14: **end if**
15: **end if**
16: **if** (There is an incoming result from p) **then**
17: Receive s_{slave} from p
18: **if** ($s_{\text{master}} \neq s_{\text{slave}}$) **then**
19: Receive the result from p.
20: **if** (terminate condition reached) **then**
21: $t_{\text{busy}} \longleftarrow$ current time.
22: Reset timer.
23: **end if**
24: **end if**
25: **end if**
26: **end while**
27: $s_{\text{master}} = s_{\text{master}} + 1$.

each column generation iteration. Now, if a slave realizes that a new request belongs to a new iteration or new branch-and-bound node, it must re-execute the preprocessing.

6.3 Computational Results

The testing environment is still the same one used in the previous chapters. Due to lack of space, Only the results for 2 test sets ("vncpp" and "vircppl") are presented. Vietnam Airlines problems are quite small and require no branching. The "vircpp" set has a similar structure as that of the "vircppl" set.

In Figure 6.2, we can see the speedup obtained from the computation of the branch-and-price code implemented in Chapter 4. But now, the parallel pricing module is used to solve the pricing subproblem. The domination of the pricing time in the sequential runs becomes a good target for parallelization. The speedup is quite high, even using

up to 32 slaves. The parallel pricing with many slaves is applicable for larger problems. Although we cannot obtain an ideal speedup, the parallel pricing helps a lot to reduce the computation time which can be seen in the last column of Table 6.1. Test problems which took several hours to be solved (see Chapter 4) are proven optimal within 20 minutes by the same branch-and-price code, but with the parallel pricing.

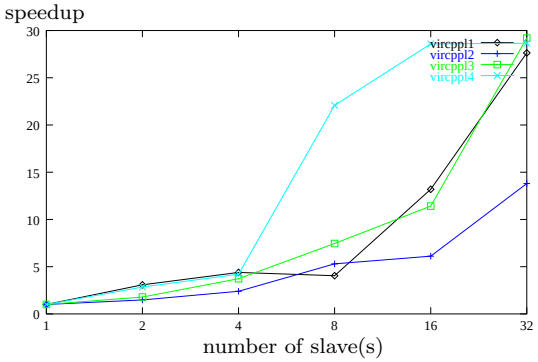


Figure 6.2: The speedups of the branch-and-price code with the standard column generation method

In order to examine the performance in more detail, the column "$\Delta t_{\text{SlavePri}}$" of the table shows the maximum difference of the pricing time among slaves, but in percent of the average slave pricing time. With 32 slaves, this number is quite high which means the load has not been well balanced. This is mainly due to the fact that the size of the problems is quite small compared to the number of processors in use. We also see the large idle time of slaves in the column "$\%_{\text{SlavePri}}$". This column denotes the average percentage of pricing time on all slaves. However, the values in this column decline approximately along with those in the column "$\%t_{\text{Pri}}$". In other words, the parallel pricing has been working efficiently. Sometimes, we can have $\%_{\text{SlavePri}} > \%t_{\text{Pri}}$ (e.g., "vircppl3"). This is explained by the early stopping technique mentioned before.

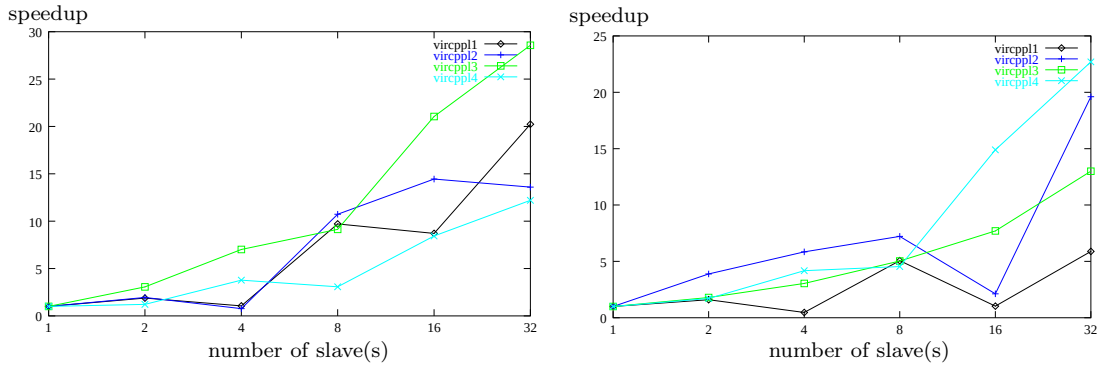
However, the early stopping technique is not very helpful in the early column generation iterations when it is extremely easy to find desired pairings. In the early iterations, a preprocessing step (e.g., constructing the network, setting weight values to the edges) is time consuming in comparison with the fast time to generate pairings. Moreover, the preprocessing has to called repeatedly. At the intermediate or final iterations, the waiting time of the master and the computing time of slaves dominate the performance (see Figure 6.1). The fast communication of HELICS introduces little overhead on the overall performance.

Likewise, the parallel pricing module is applied to the branch-and-price implementations with the stabilized column generation methods. The speedups presented in Figure 6.3 prove that the pricing method is also helpful. In Table 6.1, readers can see "t_{total}" is small in many cases. However, in comparison with the standard method, we cannot obtain a good performance. The table shows us low values of "$\%_{\text{SlavePri}}$". Observing the progress of a parallel run, one sees that the low performance relates to

Name	$\overline{z}$	B&B	Lp	n^+	n^-	Heu	time								
							$\%_{\text{Pre}}$	$\%_{\text{Pri}}$	$\%_{\text{Heu}}$	$\%_{\text{Bra}}$	$\%_{\text{Lp}}$	t_{Pri}/ite	$\%_{\text{SlavePri}}$	$\Delta t_{\text{SlavePri}}$	t_{total}
Branch-and-price with the standard column generation															
vncpp1	500943	19	231	1856	4	22	0.01	25.10	0.07	0.35	34.74	0.08	21.98	20.57	0:01:12
vncpp2	697445	9	220	1746	0	10	0.00	37.66	0.05	0.18	43.59	0.09	36.18	11.90	0:00:54
vncpp3	494154	239	889	3332	0	32	0.00	22.62	0.01	0.45	21.61	0.18	21.45	4.57	0:11:52
vncpp4	506006	147	913	3110	7	18	0.00	28.91	0.01	0.33	26.47	0.18	26.94	4.80	0:09:18
vircppl1	139344	27	426	3137	0	8	0.00	60.18	0.02	0.10	14.39	0.83	56.31	4.09	0:09:48
vircppl2	173538	25	481	3101	0	12	0.00	58.07	0.02	0.08	13.39	0.77	51.43	4.27	0:10:41
vircppl3	179818	1	264	2513	0	39	0.00	64.47	0.27	0.00	34.46	0.23	67.64	18.73	0:01:35
vircppl4	169852	21	433	3305	0	14	0.00	53.15	0.04	0.10	19.77	0.52	52.74	5.30	0:07:07
Branch-and-price with the second sliding BoxStep, $\delta B = 128$															
vncpp1	500943	9	177	537	2	0	0.40	68.03	0.00	0.17	13.78	0.18	53.91	16.40	0:00:47
vncpp2	697445	11	227	586	0	0	0.22	68.15	0.00	0.28	12.49	0.20	59.18	10.78	0:01:08
vncpp3	494154	415	1973	2638	0	0	0.20	46.38	0.00	0.53	12.07	0.26	38.19	5.88	0:18:06
vncpp4	506006	65	601	963	0	0	0.20	49.02	0.00	0.39	14.39	0.20	39.18	5.26	0:04:08
vircppl1	139344	29	319	1239	0	0	0.10	59.89	0.00	0.15	8.28	0.79	44.14	4.93	0:06:59
vircppl2	173538	27	317	1104	0	0	0.08	63.36	0.00	0.09	6.63	1.15	50.19	3.59	0:09:34
vircppl3	179818	1	146	947	0	0	0.33	81.20	0.00	0.00	17.28	0.31	59.88	10.13	0:00:56
vircppl4	169852	15	165	910	0	0	0.13	62.49	0.00	0.11	5.99	0.77	44.81	11.72	0:03:24
Branch-and-price with the stationary BoxStep, $\Delta\delta = 64$															
vncpp1	500943	21	173	623	0	0	0.17	53.02	0.00	0.47	13.71	0.18	44.54	6.10	0:01:00
vncpp2	697445	17	215	649	0	0	0.23	59.48	0.00	0.36	14.83	0.19	52.69	4.18	0:01:09
vncpp3	494154	633	3228	4713	5	0	0.16	42.33	0.00	0.47	11.60	0.25	35.92	5.75	0:31:56
vncpp4	506006	163	1140	1502	3	0	0.22	50.06	0.00	0.46	14.76	0.23	39.42	9.19	0:08:44
vircppl1	139344	51	537	1796	0	0	0.07	69.44	0.00	0.13	8.20	1.44	56.73	2.96	0:18:31
vircppl2	173538	33	331	1232	0	0	0.08	62.36	0.00	0.13	8.71	1.12	48.08	5.58	0:09:54
vircppl3	179818	3	124	966	0	0	0.46	61.08	0.00	0.08	17.66	0.24	38.07	13.52	0:00:48
vircppl4	169852	11	176	1116	0	0	0.11	61.70	0.00	0.11	9.81	0.59	47.06	7.04	0:02:48

Table 6.1: The branch-and-price code with the parallel pricing, using 32 slaves

the number of times a BoxStep method cannot find any desired pairing. At that time, the stabilized methods change the model and this results in the fast computation of pairings which has been discussed in the previous paragraph.



(a) Sliding BoxStep

(b) Stationary BoxStep

Figure 6.3: The speedups of the branch-and-price code with the stabilized column generation methods

Although there exist several problems which still need improving, the implemented parallel pricing shows to be applicable to real world large scale crew pairing problems. Furthermore, due to its design, we can easily embed a user-designed pricing method which solves a unit pricing subproblem. If one is interested in using constraint logic programming to deal with complicated rules, but afraid of the poor performance of CLP engines, the parallel pricing framework is helpful (see Hoai et al., 2003).

Chapter 7

Conclusions

The thesis presents an intensive work on the crew pairing problem. The philosophy of the thesis revolves around the application of sophisticated and new methods into the process of investigating the problem. There are many research directions in this area, but the work is only dedicated to considering branch-and-bound based methods and related algorithms and techniques.

Although having been attacked over several decades, the problem is still challenging researchers. The thesis has considered the difficulties of the problem by carefully investigating its characteristics. Despite of looking like an assignment problem, the crew pairing problem has an inherent non-linear cost structure and is constrained by non-linear airlines rules. The well-known approach for this kind of difficulty is to transfer the problem to the set partitioning model. All non-linearities have been relaxed by the enumeration. The widely-used methods are reviewed in the contexts of research and practice. As mentioned before, it can be seen that the branch-and-bound based methods are more widely used in this area. That is the reason why this work follows this approach. In order to test the application of the methods, the thesis includes a case study which would be the main problem to be attacked by implementations. Since Vietnam Airline problems of the case study are quite small, the additional problems have been randomly generated using the same properties of the case study. This is also the reason why the approaches in the thesis were preferred not to be problem-specific.

Starting the methodological parts, a typical branch-and-cut was considered, using the well-known theory in solving set partitioning problems. The thesis only focuses on points which were implemented in the implementation. They are chosen due to their effectiveness to deal with large scale set partitioning problems. Then, the code was used to solve several sets of test problems, including a well-known set (in OR community) of set partitioning problems, the set of Vietnam Airlines problems, and the set of randomly generated problems. Although there are not many new things presented in this chapter, it was helpful to show the disadvantage of the approach to deal with large scale crew pairing problems which are transferred to huge constraint matrices. This explains why the approach is only applied to small crew pairing problems or used as a solver in a heuristic method which generates only partial subproblems for

the solver. Besides the cutting generation phase, the computational difficulties of the preprocessing and linear relaxation steps are also mentioned. It is an open problem to consider other storing or solving methods in these steps. However, the branch-and-cut implementation is implemented carefully in order to use in other parts of the thesis. All the well-known test problems are solved in a reasonable computation time.

Knowing the problem in the previous chapter, the branch-and-price approach separates the master problem into two subproblems which are both solvable by computer and storable in computer memory. The whole constraint matrix is not generated in the beginning. Instead, its columns are priced out on demand from a so-called pricing subproblem. The work focuses on many methods solving this subproblem which often involves more than 90 percent of total computation time. The thesis has considered all categories of methods for it in order to choose a good one to solve the considered crew pairing problems. Note that, in a particular case, other methods could be better. However, the chosen pricing method is useful in terms of performance and extension. Then, the branch-and-price implementation solves all test problems to optimality using the same computing environment as before. This could not be done by the branch-and-cut code.

Cutting plane generation in branch-and-cut is considered dual to the process of generating columns in branch-and-price. Observing the process of producing columns, the oscillation in movement of the dual point to move to final position has been experienced. This is an inherent property of column generation methods. Stabilized column generation methods are used as a device to overcome the zig-zag movement. Although the stabilized devices have been presented in some other researches, they were considered briefly in the context of the crew pairing problem. The thesis discussed many aspects of the stabilized methods, especially the parameter control. The work also shows that, with good values of the parameters, the methods help to significantly reduce the computation time involved in solving the root node of a branch-and-bound tree. They are brought into solving the crew pairing problems. The obtained performance shows that the stabilized methods are promising to very large problems. However, it should be nice if we can find mathematical formulations to control the parameters of the methods.

The next two chapters of the thesis follow a different approach which uses the computing power resource of parallel systems. In the previous chapters, readers can see that solving the large scale test problems is very time-consuming. This is an obstacle to apply the implemented methods to real world applications. Since the airline industry is more and more dynamic, the solution of crew pairing problems should be computed more quickly. Therefore, the old parallel ABACUS is redesigned to employ a new and widely-used communication technology. Non-blocking communication is the main idea to reduce the overhead of the parallel framework. With the MPI-based parallel ABACUS, users can easily export their sequential branch-and-cut codes to parallel ones and run them on many parallel computers. To underline that, the thesis presents a process of parallelizing the sequential set partitioning solver of Chapter 3. The performance of the parallel solver proves a good design and implementation of the parallel ABACUS. The parallel set partitioning solver has good speedups in solving

large problems, especially in the branch-and-bound mode. Because it is a general framework, there are still many interesting open problems to be investigated, such as load balancing, processor grouping, local constraints/variables.

Although being parallelized, the branch-and-cut still did not overcome its main problem of memory. It is only useful with hard problems having a sufficiently small constraint matrix to be kept in computer memory. The last attempt of the thesis describes a method to parallelize the chosen pricing algorithm. With a large portion of the computation time involved in the pricing subproblem, one can expect a good speedup in parallel. With advanced techniques in parallel computing, the final code shows us a good speedup, and at the same time, solves all test problems within a small computation time. The parallel pricing is implemented in a way that facilitate future extensions and generalizations.

Finally, the thesis has performed a study on the crew pairing problem and its well-known methods. New techniques have been employed to speedup the solution process, that is the most important objective of the thesis. At the end, the implementations can solve the large crew pairing problems within minutes. A new parallel ABACUS library has been written to support the research in combinatorial optimization. The different considered methods can be combined in a heuristic to solve real-world crew pairing problems. More future research in this direction will be considered. An application of the methods to large problems of an airline also is an interesting problem in future research.

Bibliography

P. Alefragis, C. Goumopoulos, E. Housos, P. Sanders, T. Takkula, and D. Wedelin. Parallel Crew Scheduling in PAROS. In *Proceedings of Euro-Par'98 Parallel Processing: 4th International Euro-Par Conference*, 1998.

P. Alefragis, P. Sanders, T. Takkula, and D. Wedelin. Parallel Integer Optimization for Crew Scheduling. *Annals of Operations Research*, 2000. to appear.

G. Amdahl. Validity of the Single-Processor Approach to Achieving Large-Scale Computing Requirements. *Computer Design*, 6(12):39–40, 1967.

R. Anbil, E. Gelman, B. Patty, and R. Tanga. Recent Advances in Crew-pairing Optimization at American Airlines. *Interfaces*, 21(1):62–74, 1991a.

R. Anbil, E. Johnson, and R. Tanga. A Global Approach to Crew Pairing Optimization. Technical report, IBM Research Division, 1991b.

E. Andersson, E. Housos, N. Kohl, and D. Wedelin. *Operations Research in the Airline Industry*, chapter Crew Pairing Optimization. Kluwer Scientific Publishers, 1997.

B. Awerbach and R. S. Gallager. Communication Complexity of Distributed Shortest Path Algorithms. Technical report, MIT, 1985.

E. Baker, L. Bodin, W. Finnegan, and E. Ponder. Efficient Heuristic Solutions to an Airline Crew Scheduling Problem. *AIIE Transactions*, 11:79–85, 1979.

E. Balas, S. Ceria, and G. Cornuéjols. A Lift-and-Project Cutting Plane Algorithm for Mixed Zero-One Programs. *Mathematical Programming*, 58:295–324, 1993.

E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory Cuts Revisited. *Operations Research Letters*, 19:1–9, 1996.

E. Balas and M. Padberg. Set Partitioning: A Survey. *SIAM Review*, 18(4):710–760, 1976.

F. Barahona and R. Anbil. The Volume Algorithm: Producing Primal Solutions with a Subgradient Method. *Mathematical Programming*, 87(3):385–399, 2000.

E. Barnes, V. Chen, B. Gopalakrishnan, and E. Johnson. A Least-Squares Primal-Dual Algorithm for Solving Linear Programming Problems. *Operations Research Letters*, 30:289–294, 2002.

C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column Generation for Solving Huge Integer Programs. *Operations Research*, 46:316–329, 1998.

C. Barnhart, E. L. Johnson, G. L. Nemhauser, and P. H. Vance. Airline Crew Scheduling: A New Formulation and Decomposition Algorithm. *Operations Research*, 45(2): 188–200, 1997.

H. Ben Amor and Jacques Desrosiers. A Proximal Trust-Region Algorithm for Column Generation Stabilization. Technical Report G-2003-43, Les Cahiers du GERAD, 2003.

R. E. Bixby, W. Cook, A. Cox, and E. K. Lee. Parallel Mixed Integer Programming. Technical Report CRPC–TR95554, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, 1995.

M. Böhm. Parallel ABACUS - Introduction and Tutorial. Technical report, University of Cologne, 1999.

R. Borndörfer. *Aspects of Set Packing, Partitioning, and Covering*. PhD thesis, Technischen Universität Berlin, 1998.

R. Borndörfer, M. Grötschel, and A. Löbel. Scheduling Duties by Adaptive Column Generation. Technical Report ZIB-Report 01-02, ZIB, 2001.

R. Carraghan and P. M. Pardalos. An Exact Algorithm for the Maximum Clique Problem. *Operations Research Letters*, 9:375–382, 1990.

L. Cavique, C. Rego, and I. Themido. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, chapter New Heuristic Algorithms for the Crew Scheduling Problem, pages 37–47. Kluwer Academic Publishers, 1999.

Q. Chen and M. C. Ferris. FATCOP: a Fault Tolerant Condor-PVM Mixed Integer Program Solver. Technical report, Computer Sciences Department, University of Wisconsin, Madison, March 1999.

H. D. Chu, E. Gelman, and E. Johnson. Solving Large Scale Crew Scheduling Problems. *European Journal of Operational Research*, 97:260–268, 1997.

P. Chu and J. Beasley. A Genetic Algorithm for the Set partitioning Problem. Technical report, Imperial College, London, 1995.

V. Chvátal. On Certain Polytopes Associated with Graphs. *Journal of Combinatorial Theory (B)*, 18:138–154, 1975.

V. Chvátal. A Greedy Heuristic for the Set Covering Problem. *Mathematics of Operations Research*, 4:233–235, 1979.

T. F. Coleman, J. Czyzyk, C. Sun, M. Wagner, and S. J. Wright. pPCx: Parallel Software for Linear Programming. Technical Report TR 96–14, Computer Science Department, Cornell University, Ithaca, NY 14853, USA, December 1996.

CPLEX. Using the CPLEX callable library. Manual, 1997. CPLEX Optimization, Inc.

G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, New Jersey, 1963.

G. B. Dantzig and P. Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8:101–111, 1960.

G. Desaulniers, J. Desrosiers, Y. Dumas, S. Marc, B. Rioux, M. Solomon, and F. Soumis. Crew Pairing at Air France. *European Journal of Operational Research*, 97:245–259, 1997a.

G. Desaulniers, J. Desrosiers, Y. Dumas, M. Solomon, and F. Soumis. Daily Aircraft Routing and Scheduling. *Management Science*, 43:841–855, 1997b.

M. Desrochers and F. Soumis. A Reoptimization Algorithm for the Shortest Ppath Problem with Time Window. *European Journal of Operational Research*, 35:242–254, 1988.

J. Desrosiers, Y. Dumas, M. M. Solomon, and F. Soumis. *Handbooks in Operations Research and Management Science*, volume 8, chapter Time Constrained Routing and Scheduling, pages 35–139. North-Holland, 1995.

J. Desrosiers, F. Soumis, and M. Desrochers. Routing with Time Windows by Column Generation. *Networks*, 14:545–565, 1984.

O. du Merle, D. Villeneuve, J. Desrosiers, and P. Hansen. Stabilized Column Generation. *Discrete Mathematics*, 194:229–237, 1999.

J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: an Object-Oriented Framework for Parallel Branch and Bound. Technical Report RRR 40-2000, RUTCOR, Rutgers University, New Brunswick, NJ, August 2000.

J. Edmonds. Maximum Matching and a Polyhedron with 0, 1 Vertices. *Journal of Research National Bureau of Standards*, 69B:125–130, 1965.

D. Eppstein. Finding the k Shortest Paths. *SIAM Journal on Computing*, 28(2): 652–673, 1998.

R. Euler, M. Jünger, and G. Reinelt. Generalizations of Cliques, Odd Cycles and Anticycles and their Relation to Independence System Polyhedra. *Mathematics of Operations Research*, 12:451–462, 1987.

T. Fahle, U. Junker S. E. Karisch, N. Kohl, M. Sellmann, and B. Vaaben. Constraint Programming Based Column Generation for Crew Assignment. Technical report, Carmen Systems, 1999.

M. Fisher and P. Kedia. Optimal Solution of Set Covering/Partitioning Problems Using Dual Heuristics. *Management Science*, 36:674–688, 1990.

L. R. Ford and D. R. Fulkerson. Network Flow and Systems of Representatives. *Canadian Journal of Mathematics*, 10:78–84, 1958.

I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

D. Fulkerson. Blocking and Anti-blocking Pairs of Polyhedra. *Mathematical Programming*, 1:168–194, 1971.

T. Gallai. Über extreme punkt- und kantenmengen. *Ann. Univ. Sci. Budapest, Eötvös, Sect. Math.*, 2:133–138, 1959.

M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, 1979.

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manachek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachussetts, 1994.

A. M. Geoffrion. Lagrangian Relaxation for Integer Programming. *Mathematical Programming Study*, 2:82–114, 1974.

R. Gerbracht. A New Algorithm for Very Large Crew Pairing Problems. In *18th AGIFORS Symposium*, Vancouver, British Columbia, CA, 1978.

I. Gershkoff. Optimizing Flight Crew Schedules. *Interfaces*, 19:29–43, 1989.

P. C. Gilmore and R. E. Gomory. A Linear Programming Approach to the Cutting Stock Problem. *Operations Research*, 9:849–859, 1961.

F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Operations Research*, 13:533–549, 1986.

F. Glover. Tabu Search – Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

F. Glover. Tabu Search – Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

J. L. Goffin and J. P. Vial. Cutting Planes and Column Generation Techniques with the Projective Agorithm. *Journal of Optimization Theory and Applications*, 65:409–429, 1990.

D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*, pages 1–24. Addison-Wesley, 1989.

D. Goldfarb and M. Todd. *Handbooks in Operations Research and Management Science*, volume 1, chapter Linear Programming, pages 73–170. North-Holland, 1989.

G. W. Graves, R. D. McBride, and I. Gershkoff. Flight Crew Scheduling. *Management Science*, 39(6):736–745, 1993.

M. Grötschel, L. Lovász, and A. Schrijver. The Ellipsoid Method and its Consequences in Combinatorial Optimization. *Combinatorica*, 1(2):169–197, 1981.

M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer, Berlin, 1988.

N. Guerinik and M. V. Caneghem. Solving Crew Scheduling Problems by Constraint Programming. In *Proceedings of the 1st Conference of Principles and Practice of Constraint Programming*, pages 481–498, 1995.

T. Gustafsson. *A Heuristic Approach to Column Generation for Airline Crew Scheduling*. PhD thesis, Chalmers University of Technology, 1999.

G. Handler and I. Zang. A Dual Algorithm for the Constrained Shortest Path Problem. *Networks*, 10:293–310, 1980.

J. B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms*. Springer-Verlag, Berlin, 1993.

T. V. Hoai, G. Reinelt, and H. G. Bock. A Parallel Approach to the Pricing Step in Crew Scheduling Problems. In *Operations Research Proceedings 2003*, pages 165–172. Springer, 2003.

K. Hoffman and M. Padberg. Improving LP-representation of Zero-One Linear Programs for Branch-and-Cut. *ORSA Journal of Computing*, 3:121–134, 1991.

K. Hoffman and M. Padberg. Solving Airline Crew Scheduling Problems by Branch-and-Cut. *Management Science*, 39:657–682, 1993.

J. H. Holland. *Adaptation in Natural and Artifical Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

J. Hu and E. L. Johnson. Computational Results with a Primal-Dual Subproblem Simplex Method - A New Formulation and Decomposition Algorithm. *Operations Research Letters*, 25:149–157, 1999.

K. Hwang. *Computer Architecture and Parallel Computing*. McGraw Hill, New York, NY, 1984.

K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. WCB/McGraw-Hill, New York, 1998.

V. M. Jimenéz and Andrés Marzal. Computing the K Shortest Paths: a New Algorithm and an Experimental Comparison. Technical report, Departamento de Informática, Universitat Jaume I, 1999.

J. Kallrath and J. M. Wilson. *Business Optimization*. MacMillan Business, London, 1997.

J. E. Kelley. The Cutting Plane Method for Solving Convex Program. *Journal of SIAM*, 8:703–712, 1960.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimisation by Simulated Annealing. *Science*, 220:671–680, 1983.

D. Klabjan, E. Johnson, G. Nemhauser, E. Gelman, and S. Ramaswamy. Airline Crew Scheduling with Regularity. *Transportation Science*, 35(4):359–374, 2001a.

D. Klabjan, E. Johnson, G. Nemhauser, E. Gelman, and S. Ramaswamy. Solving Large Airline Crew Scheduling Problems: Random Pairing Generation and Strong Branching. *Computational Optimization and Applications*, 20(1):73–91, 2001b.

D. Klabjan, E. Johnson, G. Nemhauser, E. Gelman, and S. Ramaswamy. Airline Crew Scheduling with Time Windows and Plane-Count Constraints. *Transportation Science*, 36(3):337–348, 2002.

D. Klabjan and K. Schwan. Airline Crew Pairing Generation in Parallel. Technical report, The Logistics Institute, Georgia institute of Technology, 1999.

V. Klee and G. J. Minty. How Good is the Simplex Algorithm ? In Shisha O., editor, *Inequalities – III*, pages 159–175. Academic Press, 1972.

M. Lagerholm, C. Peterson, and B. Söderberg. Airline Crew Scheduling Using Potts Mean Field Techniques. *European Journal of Operational Research*, 120:81–969, 2000.

M. Laurent. A Generalization of Antiwebs to Independence Systems and their Canonical Facets. *Mathematical Programming*, 45:97–108, 1989.

S. Lavoie, M. Minoux, and E. Odier. A New Approach for Crew Pairing Problems by Column Generation with an Application to Air Transportation. *European Journal of Operational Research*, 35:45–58, 1988.

C. Lemaréchal. Lagrangian relaxation. In *Computational Combinatorial Optimization*, volume 2241 of *Lecture Notes in Computer Science*, pages 112–156. Springer, 2001.

C. E. Lemke, H. M. Salkin, and K. Spielberg. Set Covering by Single Branch Enumeration with Linear Programming Subproblems. *Operations Research*, 19:998–1022, 1971.

D. Levine. Application of a Hybrid Genetic Algorithm to Airline Crew Scheduling. *Computers & Operations Research*, 23(6):547–558, 1996.

J. T. Linderoth, E. K. Lee, and M. W. P. Savelsbergh. A Parallel, Linear Programming-based Heuristic for Large-Scale Set Partitioning Problems. *INFORMS Journal on Computing*, 13(3):191–209, 2001.

J. T. Linderoth and M. W. P. Savelsbergh. A Computational Study of Branch-and-Bound Search Strategies for Mixed Integer Programming. *INFORMS Journal on Computing*, 11:173–187, 1999.

R. E. Marsten, W. W. Hogan, and J. W. Blankenship. The Boxstep Method for Large-Scale Optimization. *Operations Research*, 23(3):389–405, 1975.

F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2(3):161–175, 1987.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, Knoxville, TN, June 1995.

N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equations of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21: 1087–1091, 1953.

P. J. Neame. *Nonsmooth Dual Methods in Integer Programming*. PhD thesis, The University of Melbourne, 1999.

G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

G. L. Nemhauser and L. E. Trotter. Properties of Vertex Packing and Independence System Polyhedra. *Mathematical Programming*, 6:48–61, 1973.

G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1988.

J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 1999.

M. Padberg. On the Facial Structure of the Set Packing Polyhedra. *Mathematical Programming*, 5:199–216, 1973.

M. Padberg and G. Rinaldi. A Branch-and-Cut Algorithm for the Resolution of Large Scale Symmetric Traveling Saleman Problems. *SIAM Review*, 33(1):60–100, 1991.

C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization (Algorithms and Complexity)*. Prentice-Hall, New Jersey, 1998.

T. K. Ralphs. *SYMPHONY Version 2.8 User's Guide*, 2001.

T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Parallel Branch, Cut, and Price for Large Scale Discrete Optimization. Technical report, Department of Mathematical Sciences, IBM, 2003.

C. Rego. A Subpath Ejection Chain Method for Vehicle Routing. *Management Science*, 44:1447–1459, 1998.

S. H. Roosta. *Parallel Processing and Parallel Algorithms*. Springer, New York, Berlin, Heidelberg, 1999. ISBN 0-387-98716-9.

J. Rubin. A Technique for the Solution of Massive Set Covering Problems with Applications to Airline Crew Scheduling. *Transportation Science*, 7:34–48, 1973.

E. Ruppert. Finding the k Shortest Paths in Parallel. *Algorithmica*, 28:242–254, 2000.

R. Rushmeier, K Hoffman, and M. Padberg. Recent Advances in Exact Optimization of Airline Scheduling Problems. Technical report, George Mason University, 1995.

D. M. Ryan and B. A. Foster. An Integer Programming Approach to Scheduling. In *Computer Scheduling of Public Transport*, pages 269–280. North-Holland, Amsterdam, 1981.

A. Sassano. On the Facial Structure of the Set Covering Polytope. *Mathematical Programming*, 44:181–202, 1989.

M .W. P. Savelsbergh. Preprocessing and Probing for Mixed Integer Programming Problems. *ORSA Journal of Computing*, 6:445–454, 1994.

A. J. Schaefer. *Airline Crew Scheduling under Uncertainty*. PhD thesis, Georgia Institute of Technology, 2000.

D. Schmidt. An Architectural Overview of the ACE Framework: A Case-study of Successful Cross-platform Systems Software Reuse. *USENIX login magazine, Tools special issue*, 1998.

R. Subramanian, R. Scheff, J. Quillinan, S. Wiper, and R. Marsten. Coldstart: Fleet Assignment at Delta Airlines. *Interfaces*, 24:104–120, 1994.

S. Thienel. *ABACUS - A Branch-And-CUt System*. PhD thesis, University of Cologne, 1995.

S. Thienel. *ABACUS 2.0: User's Guide and Reference Manual*. University of Cologne, 1997.

L. E. Trotter. Solution Characteristics and Algorithms for the Vertex Packing problem. Technical Report 168, Dept. of Operations Research, Cornell University, Ithaca, NY, 1973.

L. E. Trotter. A Class of Facet Producing Graphs for Vertex Packing Polyhedra. *Discrete Mathematics*, 12:373–388, 1975.

N. H. Tung. Một mô hình lập lịch thông minh giải bài toán điều độ trong hàng không. Master's thesis, Ho Chi Minh City University of Natural Science, 1998.

P. H. Vance, A. Atamtürk, C. Barnhart, E. Gelman, E. L. Johnson, A. Krishna, D. Mahidhara, G. L. Nemhauser, and R. Rebello. A Heuristic Branch-and-Price Approach for the Airline Crew Pairing Problem. Technical report, Georgia Institute of Technology, 1997.

F. Vanderbeck. *Decomposition and Column Generation for Integer Programs*. PhD thesis, Universite Catholique de Louvain, Belgium, 1994.

R. J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, Boston, 2001.

Vietnam Airlines. Flight Operations Manual: Flight Time Limitations and Rest Periods, 1998. Chapter 7.

D. Wedelin. An Algorithm for Large Scale 0-1 Integer Programming with Applications to Airline Crew Scheduling. *Annals of Operations Research*, 57:283–301, 1995.

L. A. Wolsey. *Integer Programming*. John Wiley, New York, 1998.

G. Yarmish. *A Distributed Implementation of the Simplex Method*. PhD thesis, Polytechnic University, New York, USA, 2001.

J. W. Yen. *A Stochastic Programming Formulation of the Stochastic Crew Scheduling Problem*. PhD thesis, University of Michigan, 2000.

T. H. Yunes, A. V. Moura, and C. C. de Souza. *Practical Aspects of Declarative Languages*, volume 1753 of *Lecture notes in computer science*, chapter A Hybrid Approach for Solving Large Scale Crew Scheduling Problems, pages 293–307. Springer, 2000.

Index