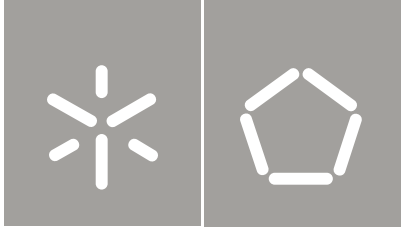
**Universidade do Minho**
Escola de Engenharia

Filipe Pereira Pinto da Cunha e Alvelos

# Branch-and-Price and Multicommodity Flows

Fevereiro de 2005



**Universidade do Minho**
Escola de Engenharia

Filipe Pereira Pinto da Cunha e Alvelos

# Branch-and-Price and Multicommodity Flows

Dissertação submetida à Universidade do Minho para a obtenção do grau de Doutor no Ramo de Engenharia de Produção e Sistemas, área de Investigação Operacional

Trabalho efectuado sob a orientação de

**Professor Doutor José Manuel Vasconcelos Valério de Carvalho**
Departamento de Produção e Sistemas da Escola de Engenharia da Universidade do Minho

Fevereiro de 2005

To the Memory of my Father.

To my Mother.

To their grandson, Miguel.

# Abstract

In this Thesis, we address column generation based methods for linear and integer programming and apply them to three multicommodity flow problems.

For (mixed) integer programming problems, the approach taken consists in reformulating an original model, using the Dantzig-Wolfe decomposition principle, and then combining column generation with branch-and-bound (branch-and-price) in order to obtain optimal solutions. The main issue when developing a branch-and-price algorithm is the branching scheme. The approach explored in this work is to branch on the variables of the original model, keeping the structure of the subproblems of the column generation method unchanged. The incorporation of cuts (branch-and-price-and-cut), again without changing the structure of the subproblem, is also explored.

Based on that general methodology, we developed a set of *C++* classes (*ADDing – Automatic Dantzig-Wolfe Decomposition for INteger column Generation*), which implements a branch-and-price algorithm. Its main distinctive feature is that it can be used as a "black-box": all the user is required to do is to provide the original model. *ADDing* can also be customised to meet a specific problem, if the user is willing to provide a subproblem solver and/or specific branching schemes.

We developed column generation based algorithms for three multicommodity flow problems. In this type of problems, it is desired to route a set of commodities through a capacitated network at a minimum cost.

In the linear problem, each unit of each commodity is divisible. By using a model with variables associated with paths and circuits, we obtained significant improvements on the solution times over the standard column generation approach, for instances defined in planar networks (in several instances the relative improvement was greater than 60%).

In the integer problem, each unit of each commodity is indivisible; the flow of a commodity can be split between different paths, but the flow on each of those paths must be integer. In general, the proposed branch-and-price algorithm was more efficient than Cplex 6.6 in the sets of instances where each commodity is defined by an origin-destination pair; for some of the other sets of instances, Cplex 6.6 gave better time results.

In the binary problem, all the flow of each commodity must be routed along a single path. We developed a branch-and-price algorithm based on a knapsack decomposition and modified (by using a different branching scheme) a previously described branch-and-price-and-cut algorithm based on a path decomposition. The outcome of the computational tests was surprising, given that it is usually assumed that specific methods are more efficient than general ones. For the instances tested, a state-of-the-art general-purpose (Cplex 8.1) gave, in general, much better results than both decomposition approaches.

# Resumo

Nesta Tese, abordam-se métodos de geração de colunas para programação linear e inteira. A sua aplicação é feita em três problemas de fluxo multicomodidade.

Para problemas de programação inteira (mista), a abordagem seguida é a de reformular um modelo original, utilizando o princípio de decomposição de Dantzig-Wolfe, e combinar geração de colunas com o método de partição e avaliação (partição e geração de colunas) para a obtenção de soluções óptimas. A questão essencial no desenvolvimento de um algoritmo deste tipo é a estratégia de partição. A abordagem seguida neste trabalho é a de realizar a partição nas variáveis do modelo original, mantendo a estrutura do subproblema do método de geração de colunas. A incorporação de cortes, ainda sem alteração da estrutura do subproblema, é também explorada.

Com base nesta metodologia geral, foi desenvolvido um conjunto de classes em C++ (*ADDing − Automatic Dantzig-Wolfe Decomposition for INteger column Generation*), que implementa um algorithmo de partição e geração de colunas. A sua característica fundamental é apenas ser requerido ao utilizador a definição de um modelo original. Num modo mais avançado, o utilizador pode implementar algoritmos para resolver o subproblema e/ou esquemas de partição.

Foram desenvolvidos algoritmos baseados em geração de colunas para três problemas de fluxo multicomodidade. Neste tipo de problemas, pretende-se encaminhar um conjunto de comodidades através de uma rede capacitada, minimizando o custo.

No problema linear, cada unidade de cada comodidade é divisível. Utilizando um modelo com variáveis associadas a caminhos e a circuitos, obtiveram-se melhorias significativas nos tempos de resolução em relação ao método de geração de colunas usual, para instâncias definidas em redes planares (em várias instâncias a melhoria relativa foi superior a 60%).

No problema inteiro, cada unidade de cada comodidade é indivisível; o fluxo de uma comodidade pode ser dividido por diferentes caminhos, mas o fluxo em cada um deles tem de ser inteiro. Em geral, o algoritmo de partição e geração de colunas foi mais eficiente do que o *software* Cplex 6.6 nos conjuntos de instâncias em que cada comodidade é definida por um par origem-destino; para alguns dos outros conjuntos de instâncias, o *software* Cplex 6.6 obteve melhores resultados.

No problema binário, todo o fluxo de cada comodidade apenas pode utilizar um caminho. Foi desenvolvido um algoritmo de partição e geração de colunas baseado numa decomposição de mochila e modificado (através de um esquema de partição diferente) um algoritmo de partição e geração de colunas com cortes, previamente descrito, baseado numa decomposição por caminhos. Os resultados dos testes computacionais foram surpreendentes, dado que é usualmente assumido que métodos específicos são mais eficientes do que métodos gerais. Para as instâncias testadas, o *software* Cplex 8.1 obteve, em geral, resultados muito melhores do que as duas decomposições.

# Acknowledgements

I am most grateful to Professor Valério de Carvalho, who introduced me to branch-and-price algorithms with a contagious enthusiasm. His guidance and support, every time I needed them, were of inestimable value. I also deeply acknowledge his sharing of experience and knowledge. I will keep his positive attitude and good practices as examples, and his friendship as a priceless commodity.

Carina's companionship and professionalism made these years much more pleasant and productive. Acácio was a constant presence every time I needed human (and technical...) support. Vítor's sense of humour gave me the right perspective whenever I was losing it. Their friendship and encouragement were great.

Carina Pimentel contributed with programming the random instance generator used in Chapters 4 and 5 and executing some of the computational tests described in those Chapters.

Professor Antonio Frangioni shared his bundle code (and gave support on running it) allowing the comparative computational tests that are described in Chapter 5. He also shared valuable comments that helped improving that Chapter. Furthermore, the email discussions about his experience on developing code for cutting plane methods for non-differentiable optimisation were a source of learning, inspiration and motivation.

I thank the "Departamento de Produção e Sistemas", and, in particular, the "Optimização e Investigação Operacional" group, for the support given during these years.

I would like to express my gratitude to Carina, Acácio, Vítor, Sérgio, and José António for the suggestions that helped to improve the text.

My attraction for Operational Research started in the classes of Professor José Soeiro Ferreira and Professor Jorge Pinho de Sousa, when I was an undergraduate student. I thank them for being the first responsibles for the fulfilment of my professional life.

Telmo is everything I could expect from a great friend. It is difficult to express my delight for all the time that we spent together, since we were undergraduate students, in such different situations: from work sessions to all kind of discussions about almost everything. His careful reading of this work was a precious contribution.

The support of my mother-in-law, Isabel, and of my brothers-in-law, Gonçalo and Ilídio, translated into a warm feeling that made these years brighter.

As it has always happened in my life, I deeply valued the example and felt the love of my Sister and of my Brother, through these Thesis years.

The example of humanism and strength of my Mother guide my life. Her love is the most sweet encouragement.

Margarida is a dream.

This Thesis was partially supported by *Fundação para a Ciência e Tecnologia* (*Projecto POSI/1999/SRI/35568*) and *Centro de Investigação Algoritmi, Universidade do Minho*.

# Table of Contents

# 1 General Introduction

In this Chapter, we provide a brief introduction to the main topics of the present Thesis. In Section 1.1, a non-technical and historical perspective on Linear and Integer Programming, the broader field that encompasses the subjects of this Thesis, is presented. In Section 1.2, a historical perspective on the branch-and-price method is given and its present relevance is pointed out. Multicommodity flow models are briefly introduced in Section 1.3, in the broader context of network modelling. In Section 1.4, we specify the main contributions of this work. Finally, in Section 1.5, the structure of the present Thesis is described.

## 1.1 Linear and Integer Programming

For almost six decades now, Linear and Integer Programming have been major research topics in Operational Research, playing a decisive role in its definition as an applied scientific area.

Linear Programming, whose first milestone was the Simplex algorithm of G. B. Dantzig, developed in the late 1940s, deals with the optimisation of systems where different activities compete for a set of scarce resources, their relations being expressed mathematically by linear functions. Integer Programming can be considered as an extension of Linear Programming (usually credited to R. E. Gomory in the late 1950s), allowing the mathematical modelling and algorithmic treatment of a broader type of decision problems.

Nowadays, Linear and Integer Programming are established disciplines. Their methods have been successfully applied in a large number of practical problems and there are robust and efficient software implementations, enabling their use in yet more problems (whose number is virtually infinite). The "age of optimisation" (announced by G. L. Nemhauser (Nemhauser, 1994)) is already over one decade old.

However, Integer Programming is still a challenging field of research. There are no known polynomial algorithms to solve (that is, to obtain an optimal solution to) the general Integer Programming problem (as opposed to the general Linear Programming problem). Although several important specific problems have been studied for several decades, and very significant progresses have been made in their resolution, they remain difficult to solve. Some of them have a combinatorial structure, that is, the set of feasible solutions is a set of objects that can be explicitly enumerated, but their number is too large for the enumeration to be efficient in a solution procedure.

In order to tackle this inherent complexity of Integer Programming, several approaches have been developed in the last decades. R. E. Gomory was the pioneer of cutting plane methods in the late 1950s. Roughly at the same time, A. H. Land and A. G. Doig developed branch-and-bound, an implicit enumeration procedure. Those two classical methods can be taken as the basis for all the subsequent major developments in Integer Programming methods.

Although branch-and-cut has its roots in the work of H. P. Crowder, E. L. Johnson, and M. W. Padberg in the early 1980s, its potential for solving a large number of problems only began to be largely explored in the 1990s. In a very general framework, branch-and-cut can be seen as the combination of cutting planes (generally, stronger than the ones used in the original work of R. E. Gomory) and branch-and-bound.

Branch-and-price was first developed by J. Desrosiers, F. Soumis, and M. Desrochers in

the middle 1980s. It can also be seen as a combination of two existing methods: branch-and-bound and column generation. Being the subject of this work, we will refer to branch-and-price, and related concepts and methods, such as Dantzig-Wolfe decomposition and Lagrangean relaxation, in more detail later in this Introduction. For the time being, we only would like to point out that the combination of branch-and-price and branch-and-cut (branch-and-cut-and-price) has also been a major topic of research in the last decade.

It should be noted that all these methods have Linear Programming as a fundamental component, and thus every breakthrough in Linear Programming may have a significant improvement in their efficiency. A comprehensive review of the fundamental concepts of the above methods is given in (Johnson et al., 2000). Less common exact methods, not based in Linear Programming, are explored in (Aardal et al., 2002).

All the methods mentioned above were devised to obtain optimal solutions. A different approach in Integer Programming / Combinatorial Optimisation is to seek good solutions. Heuristics have been an important field of research since Operational Research techniques started to be applied in the solution of practical problems. A substantial progress in those approaches began in the late 1970s with the development of the first meta-heuristics, which are still the object of intense research by the Operational Research and neighbouring scientific communities. Approximation methods, which are devised for finding sub-optimal solutions with a guarantee of their quality, have also been the subject of research among those scientific communities.

## 1.2 Branch-and-Price

Branch-and-price amounts to the combination of column generation and branch-and-bound. Column generation is used to solve the Linear Programming problems that are used as relaxations within the branch-and-bound method.

The roots of column generation date back to the late 1950s and the early 1960s. L. R. Ford and D. R. Fulkerson for the first time solved a specific Linear Programming model (maximal multicommodity flow) not defining explicitly all its variables. That approach served as inspiration for the decomposition principle of G. B. Dantzig and P. Wolfe, which allows reformulating a general Linear Programming model in such a way that its structure (that is, its parts and the relation between them) is made clear and ready to be explored algorithmically (namely, by column generation). P. C. Gilmore and R. E. Gomory for the first time used column generation as the fundamental piece for obtaining heuristic solutions in a specific Integer Programming problem (the cutting stock problem), although not combined with branch-and-

bound. Their combination, which provides optimal solutions, was made about 20 years later by J. Desrosiers, F. Soumis, and M. Desrochers in a vehicle routing problem.

Meanwhile, a closely related approach, Lagrangean relaxation, whose first use in Integer Programming is due to M. Held and R. M. Karp, emerged in the early 1970s. The simplicity of the subgradient method and variants used in Lagrangean relaxation was a main factor to its subsequent generalisation.

The Dantzig-Wolfe decomposition principle and Lagrangean relaxation are (primal-dual) equivalent. Correspondingly, column generation and the cutting plane method for non-differentiable optimisation (from J. E. Kelley in the early 1960s) used for solving the resulting reformulated problems, are the same, with different (primal-dual) perspectives. We believe that the main reason for why only in the 1990s their potential started to be exploited and developed (as opposed to the subgradient method mentioned above) was the lack of efficient and robust software implementations of Linear Programming algorithms.

As pointed out in several surveys (Barnhart et al., 1998; Hoffman, 2000; Johnson et al., 2000; Wilhelm, 2001; Lübbecke and Desrosiers, 2002) several issues of branch-and-price methods deserve further research. In this work, we aim at contributing to the research of some of those issues, which we will refer to after a brief introduction to the other main topic of this work: multicommodity flow problems.

## 1.3 Multicommodity Flow Problems

Network models played an important role in the development of Operational Research since its origins. The classical transportation problem (one of the first problems solved in a computer in 1951) is a landmark in the history of Operational Research. The work of L. R. Ford and D. R. Fulkerson in the late 1950s and early 1960s established network flows as a major field of application and research in Linear Programming.

Network models are used to deal with a large number of decisions in our society. From electrical to water systems, from railway to communications systems, networks are everywhere. In a network flow model, the usual modelling approach is to define decision variables as the flow on each arc. Flow conservation constraints force the flow entering each node to be equal to the flow leaving that same node. Additional variables and constraints allow the consideration of a large number of different problems, arising in several application areas such as transportation/ distribution and production planning.

Multicommodity network flow models deal with problems where several different commodities share the same network. In the multicommodity flow problems studied in this work, it is intended to route all the commodities from their origins to their destinations at

minimum cost, in a network with capacitated arcs.

A Linear Programming model for those problems is composed of two sets of constraints: flow conservation of the commodities and capacities of the arcs. Such a model is a large one: it has one decision variable for each commodity and arc; one constraint for each commodity and node; and one constraint for each capacitated arc. The fact that, by neglecting the capacity constraints, a set of independent problems is obtained (one for each commodity), leads to the efficiency of decomposition methods for their solution, which has been done since the work of J. A. Tomlin in the middle 1960s.

In this work, we apply the decomposition approach just outlined combined with branch-and-bound, to the integer multicommodity flow problem where the units of the commodities cannot be split; and to the binary multicommodity flow problem where each commodity must use a single path. We also consider the linear minimum cost multicommodity flow problem defined in a planar network, for which we discuss a procedure to improve the efficiency of the column generation algorithm.

## 1.4 Contributions

In our view, the main contributions of this Thesis are the following.

A general branch-and-price methodology is explored for using Dantzig-Wolfe decomposition in (mixed) integer problems for which a compact model is known. Its main feature is the compatibility of branching rules and subproblems, allowing the incorporation of cuts. The extension of that approach for multiple Dantzig-Wolfe decomposition is proposed. We implemented that methodology in *ADDing − Automatic Dantzig-Wolfe Decomposition for INteger column Generation*, a general branch-and-price algorithm coded in C++.

In its basic use, all the user is required to do is to provide an original (mixed) integer model. *ADDing* automatically decomposes the original model and uses branch-and-price to obtain an (integer) optimal solution. In a more advanced use, the user may provide specific subproblem solvers and branching rules through a few, hopefully simple, functions.

Typically, the implementation of a branch-and-price algorithm is a time-consuming task. Two options exist: to develop specific code for the problem at hand or to use existing frameworks (such as Abacus or COIN/BCP) that require an in-depth knowledge of their internal structure but less coding effort. With *ADDing*, we intend to provide a third alternative.

Based on the branch-and-price methodology mentioned above, we propose three branch-and-price algorithms for two multicommodity flow problems.

For the integer multicommodity flow problem (MFP), the formulation based on flows in

paths is used. To our knowledge, the development of an exact decomposition algorithm for that problem is made for the first time. The branching scheme proposed is the application of the general methodology mentioned above. In the case of the integer MFP, an interesting issue arises: the subproblems in the nodes of the search tree may suggest rays, although, in an optimal solution, their weight is null. The approach proposed here to deal with that issue might be used in other network flow problems.

For the binary MFP, we explore two decompositions. The first is the decomposition based on paths previously applied to that problem by (Barnhart et al., 2000), which also uses cuts. We use a different branching scheme and present comparative computational results.

The second decomposition is based on defining the subproblem as a binary knapsack problem for each arc. Although this type of knapsack decomposition has been used in other problems, to our knowledge, this is the first time it is used for the binary MFP. Its potential advantage is that the lower bound it provides is, in general, of better quality than the one provided by the linear relaxation of the original formulation or by the path decomposition.

We present computational tests of the two branch-and-price algorithms and compare them with a general-purpose solver. Although other specific algorithms have been proposed in the literature, such a comparison is made for the first time.

In this work, the relation between Dantzig-Wolfe decomposition and Lagrangean relaxation is explicitly discussed, as is the relation between column generation and methods for solving the Lagrangean dual. The insight given by those relations is a fundamental issue when developing stabilising procedures for column generation. An application of the use of extra dual cuts (Carvalho, 2000) to a planar multicommodity flow problem is made. The approach is based on the use of a model that includes extra circuit variables, besides the usual path variables, allowing the implicit consideration of paths that are not generated by the subproblem. We present comparative computational results on the proposed approach, standard column generation, and a bundle method implementation.

## 1.5 Outline

Each Chapter of the present Thesis is essentially self-contained. Although some issues could be presented in a dependent manner, we choose to present Dantzig-Wolfe decomposition and column generation based algorithms in a general way (Chapter 2) and dedicate one Chapter to the general implementation that was carried out (Chapter 6), devoting each of the remaining Chapters to one different multicommodity flow problem (Chapters 3, 4 and 5).

In more detail the organisation of the present Thesis is as follows.

In Chapter 2, the fundamental theory of the Dantzig-Wolfe decomposition principle and column generation based methods is reviewed. We aim at providing a comprehensive overview and contextualisation of those approaches for solving Linear and Integer Programming problems, by reviewing their fundamental conceptual and algorithmic aspects and by providing references to applications, related methods and recent developments. The general approach taken encompasses the exposition of a general branching scheme for branch-and-price, incorporation of cuts (branch-and-price-and-cut), and the development of multiple Dantzig-Wolfe decomposition.

Chapters 3, 4 and 5 are devoted to the development and testing of column generation based algorithms for three different multicommodity flow problems: the (general) integer, the binary, and the linear defined in a planar network, respectively.

In Chapter 3, a branch-and-price algorithm is developed for the integer multicommodity flow problem following the general approach presented in Chapter 2. We review solution methods that have been devised for the linear relaxation of the problem. In the nodes of the search tree, we propose a formulation that includes cycle variables, in addition to the usual path variables. Computational tests of the proposed algorithm and of a general-purpose solver are presented and discussed.

In Chapter 4, we present two branch-and-price algorithms for the binary multicommodity flow problem. Based on the Dantzig-Wolfe principle, we derive two different decompositions depending on the subproblem definition. One decomposition captures the network structure of the problem and the other provides better quality lower bounds. For the decomposition based on paths, we compare the developed branching rule with one previously presented by (Barnhart et al., 2000). We present computational results for the two decompositions and compare them with the ones given by a general-purpose integer programming solver.

In Chapter 5, we propose a way of accelerating a column generation algorithm for the linear minimum cost multicommodity flow problem. We use a new model that, besides the usual variables associated with paths, has a polynomial number of extra variables (when the problem is defined in a planar network), associated with circuits. We present computational results for the comparison of this new approach with standard column generation, a bundle method, and a general-purpose solver.

In Chapter 6, we describe *ADDing – Automatic Dantzig-Wolfe Decomposition for INteger column Generation* – a general branch-and-price algorithm implementation in C++. We describe its use and internal structure, presenting further details in the Appendix. Future development directions are also discussed.

Finally, in Chapter 7 we draw the overall conclusions taken from this work and point out some directions for further research.

# 2 Dantzig-Wolfe Decomposition and Column Generation Based Algorithms

In this Chapter, we address Dantzig-Wolfe decomposition and column generation based algorithms for linear and integer programming.

We review the main theoretical aspects of Dantzig-Wolfe decomposition and column generation, as approaches for solving structured models, both in a linear programming perspective and in a Lagrangean relaxation perspective. Different alternatives for implementing a column generation algorithm are surveyed.

The use of Dantzig-Wolfe decomposition in integer programming is discussed. We detail a general branching scheme for combining column generation and branch-and-bound (branch-and-price). The extension of that combination to allow the incorporation of cuts is also detailed.

We explore multiple Dantzig-Wolfe decomposition / multiple column generation in the context of the general branching scheme exposed, which, to our best knowledge, is made here for the first time.

## 2.1 Introduction

Branch-and-price combines two well-established methods, column generation and branch-and-bound, to obtain the optimal solution of (mixed) integer problems. Although those two methods are known since the late 1950s, only in the middle 1980s was developed their first combination to obtain optimal integer solutions for a routing problem (Desrosiers et al., 1984) and only in the late 1990s the first revision paper about branch-and-price was published (Barnhart et al., 1998).

Over the last years, a renewed interest on column generation based algorithms has appeared, judging from the large number of publications on the subject (see Table 2.1, at the end of this Section, pages 12 and 13).

We believe there are two main reasons for that. Firstly, the availability of appropriate computational tools made their implementations easier and more robust, exposing their advantages over other methods of the same "family" (such as the subgradient method). Secondly, their successful application on problems with great economical and social impact, such as the ones faced by the airline and transportation industries.

Several motivations can (co)exist for developing a column generation based algorithm.

First of all, it is a decomposition algorithm. We may have a compact formulation (a model where is possible to consider all the decision variables and constraints explicitly) but so large that the possibility of solving it directly, in an efficient way, must be ruled out. Being so, a decomposition approach, where solutions to parts of the model are obtained by solving smaller (sub)problems and then combined to form a solution to the overall problem, is attractive. It is worth noting, as done, for example, in (Williams, 1999) and (Martin, 1999), that the vast majority of practical problems, for which a compact model can be devised, has some kind of structure: we can identify submodels within the model. Thus, even if it is feasible to solve the compact model, for computational memory reasons or for taking advantage of the efficient algorithms that may exist for those submodels, column generation based algorithms are an appealing approach.

The decomposition framework described in the previous paragraph can be extended to problems where the compact formulation is nonlinear. Although column generation is essentially a linear programming technique, there will be no conceptual changes if the non-linearity of the model is confined to the subproblems (this is a usual motivation for routing and scheduling applications, where the nonlinear subproblems are solved by dynamic programming).

Another motivation for the use of column generation based algorithms is that a compact formulation for the problem at hand is not known.

A last motivation is a fundamental one when dealing with integer programming models. In those models, a major issue when they are attacked by methods based on bounds given by relaxations (such as branch-and-bound) is their quality. Alternative models with a huge number of columns, under certain circumstances, give better lower bounds (considering a minimisation problem). A classical illustration is the pioneer work of Gilmore and Gomory in the cutting stock problem (Gilmore and Gomory, 1961; Gilmore and Gomory, 1963) where a column generation algorithm is devised to obtain "good" linear solutions that are then rounded by a heuristic.

In this work, we focus on the application of branch-and-price algorithms in problems where a compact formulation is known. The Dantzig-Wolfe Decomposition (DWD) principle (Dantzig and Wolfe, 1960) is used to reformulate the compact model (called original). Column generation is then used to deal with the huge number of variables of the reformulated model. In order to obtain integer solutions, branch-and-bound is used in such a way that the relaxed problems of the nodes of its search tree are solved by column generation. Using this approach, there is a guarantee that an optimal solution (with the desired accuracy) will be found. However, it can also serve as a framework for the development of heuristics. Three examples of such type of approaches are: (i) to include, in the problems solved in the nodes of the search tree, only the columns that, in its root, had a reduced cost less than a preset threshold value; (ii) to stop the search of the tree as soon as a feasible integer solution is found; (iii) to round the fractional solution obtained in the root node.

It is worth to emphasise that Dantzig-Wolfe reformulation is not the only source of models where the use of column generation based algorithms is appropriate.

In (Wilhelm, 2001) column generation based algorithms are classified in three types. Type I algorithms are based on the selection of a subset of promising variables (that is, variables that hopefully will have positive values in an optimal solution) from the huge set of existing variables. Then a master problem where only those variables are considered (thus, a restricted master problem) is solved in order to identify their best combination. Clearly, this column generation *a priori* approach does not guarantee optimality.

Type II algorithms are based on the iterative exchange of information between the (restricted) master problem and the subproblem. In every iteration, the master problem (where the variables generated by the subproblem in previous iterations are considered) is optimised, providing guidance for the subproblem to generate promising variables for the next iteration.

Type III algorithms are similar to the ones of type II, but the master problem results from a DWD. According to this classification, some approaches can be included in both type II and III, since some of the Dantzig-Wolfe reformulated models have a natural interpretation when

directly formulated. Furthermore, it has been proved that for each Dantzig-Wolfe reformulated model there is a compact original formulation (Villeneuve et al., 2003).

In this work we focus on type III column generation and its use on integer programming. This approach, where a compact model is reformulated using DWD, can be applied in every problem for which a compact formulation is known. However, decomposable models are natural candidates for this type of algorithms. Clearly, this does not mean that the application of column generation based algorithms is limited to a few problems. Besides the fact already mentioned that most practical models are decomposable, this is confirmed by the large number of problems where Lagrangean relaxation (in particular the subgradient method for solving the Lagrangean dual) was successfully applied in the last four decades. For all those problems, DWD can also be applied. In fact, despite their different origins (nonlinear programming in the case of Lagrangean relaxation, linear programming in the case of DWD), there is a dual equivalence between the two approaches. As for solution methods, the cutting plane method of Kelley (Kelley, 1960) when applied to solve the dual Lagrangean problem is dual equivalent to column generation applied to solve the DWD reformulated model.

In this primal-dual perspective, we can think of Lagrangean relaxation and DWD as the same decomposition for which different solution methods exist, being the most well-known: subgradient (which is frequently (mis)taken as a synonym of Lagrangean relaxation), bundle, volume and Kelley's cutting plane / column generation. Two distinctive features characterise column generation: its understanding can be confined to linear programming (as an extension of the simplex algorithm) and it has a natural primal interpretation. Although being technically equivalent to Kelley's cutting plane method, the column generation primal perspective has some advantages, in particular when it is the base for the solution of integer problems. Firstly, it makes the identification and the use of cuts (branch-and-price-and-cut) easier. Secondly, it also makes the incorporation of (primal) heuristics that may capture specific aspects of the problem at hand easier. Thirdly, and last, the promising hybridisation of linear/integer programming with constraint logic programming techniques certainly requires a primal perspective of the problem (as an example, see (Fahle et al., 2002)). We note that these advantages are only related with the conceptual framework for the development of decomposition algorithms. Certainly, the insight given by the dual methods/perspective plays a crucial role on column generation based algorithms. Two recent publications (Lemaréchal, 2003; Frangioni, 2004) discuss in detail the relation between DWD and Lagrangean relaxation.

A list of applications, and their references, to column generation based algorithms is given in Table 2.1. Additional references can be found in (Desrosiers et al., 1995) (a survey on time constrained routing and scheduling where branch-and-price algorithms for several of those problems are described), and (Wilhelm, 2001; Lübbecke and Desrosiers, 2002) (surveys of the

use of column generation based algorithms in integer programming).

This Chapter is organised as follows. In the next Section, the use of DWD principle and the column generation technique are described. In Section 2.3, the close relation of DWD with Lagrangean relaxation is made explicit and column generation is discussed in the broader context of methods to solve the Lagrangean dual. In Section 2.4, column generation variants and stabilisation procedures are discussed. In Section 2.5, the use of the DWD principle is extended to integer programming problems and the combination of column generation and branch-and-bound (branch-and-price) is described. In the same Section, we also develop multiple Dantzig-Wolfe decomposition / multiple column generation. Finally, in Section 2.6, we conclude this Chapter, by reviewing the main aspects discussed in it.

| Application | Reference(s) | Application | Reference(s) |
|---|---|---|---|
| Vehicle routing with time windows | (Desrosiers et al., 1984; Desrochers et al., 1992; Kohl et al., 1999) | Traffic assignment | (Ribeiro et al., 1989) |
| Vehicle scheduling | (Ribeiro and Soumis, 1994; Desaulniers et al., 1998) | Traffic equilibrium | (Larsson et al., 2004) |
| Simultaneous vehicle and crew scheduling | (Desaulniers et al., 2001; Haase et al., 2001; Freling et al., 2003) | Time slot assignment in a satellite system | (Lee and Park, 2001) |
| Pickup and delivery | (Savelsbergh and Sol, 1998; Christiansen and Fagerholt, 2002; Lübbecke and Zimmermann, 2003) | Spectrum auctions | (Günlük et al., 2002) |
| Multiple tour maximum collection | (Butt and Ryan, 1999) | Probabilistic logic | (Jaumard et al., 1991) |
| Air network design | (Barnhart and Schneur, 1996) | Coalition formation in multi-agent systems | (Tombus and Bilgiç, 2004) |
| Fleet assignment | (Hane et al., 1995) | Management of spare parts | (Mehrotra et al., 2001) |
| Crew scheduling | (Desrochers and Soumis, 1989; Vance et al., 1997; Yan and Chang, 2002; Yan et al., 2002) | Delivery planning | (Boland and Surendonk, 2001) |
| Aircrew rostering | (Gamache et al., 1999) | Assembly system design | (Wilhelm, 1999) |
| Staff scheduling | (Jaumard et al., 1998; Mehrotra et al., 2000; Sarin and Aggarwal, 2001; Bard and Purnomo, 2004; Eveborn and Rönnqvist, 2004) | Forest management | (Martins et al., 2003) |
| Job scheduling | (Akker et al., 1999; Chen and Powell, 1999; Akker et al., 2000; Akker et al., 2002) | Course registration | (Sankaran, 1995) |

| | | | |
|---|---|---|---|
| Cutting stock and bin packing | (Vance et al., 1994; Carvalho, 1999; Vanderbeck, 1999; Alves and Carvalho, 2003) | Ship routing and inventory management | (Christiansen and Nygreen, 1998) |
| Facility location | (Shaw, 1999; Klose and Drexl, 2002) | Supply chain management | (Bredström et al., 2004) |
| P-median | (Ceselli and Righini, 2002; Lorena and Senne, 2004; Senne et al., 2005) | Ring network design | (Henningsson et al., 2002) |
| Lot sizing | (Vanderbeck, 1998; Kang et al., 1999; Degraeve and Jans, 2003) | Shipment planning at oil refineries | (Persson and Göthe-Lundgren, 2005) |
| Switch-box routing | (Jørgensen and Meyling, 2002) | Sorting permutations by reversals | (Caprara et al., 2001) |
| Circuit partitioning | (Ebem-Chaime et al., 1996) | Beam-on time in cancer radiation | (Boland et al., 2004) |
| Placement of multiplexers | (Sutter et al., 1998) | Steiner tree packing | (Jeong et al., 2002) |
| Generalised assignment | (Savelsbergh, 1997) | Graph coloring | (Mehrotra and Trick, 1996) |
| Car assignment to trains | (Lingaya et al., 2002) | Maximum stable set | (Bourjolly et al., 1997) |
| Channel assignment | (Jaumard et al., 2002) | Clustering | (Mehrotra and Trick, 1998) |

Table 2.1 Applications of column generation based algorithms.

## 2.2 Dantzig-Wolfe Decomposition and Column Generation

### 2.2.1 Structured models

Large-scale optimisation problems typically have some kind of structure: it is possible to identify parts of the problem that are defined in a similar way. We give four brief examples. In a production planning problem over a temporal horizon, the decision about the quantities to produce of each product, given a set of common resources, defines a similar problem for each product. In vehicle routing, the problem of defining the route for each vehicle is similar (in the case vehicles have the same characteristics, that problem is equal for all vehicles – "which route should *I* take?"). In the generalised assignment problem, where the maximum profit assignment of a set of jobs to a set of agents with limited capacity is desired, the problem that each agent faces is similar ("which jobs should *I* make?"). In machine scheduling, where a set of tasks must be scheduled in a set of machines, the problem of each machine is similar ("which tasks should *I* perform?").

When formulated with linear/integer programming this kind of problems give rise to structured models. The nonzero coefficient values in the constraints do not appear in random places, but in blocks, as exemplified in Figure 2.1, where the senses and right-hand-sides of the

constraints are omitted. Assuming that blocks *F* and *E* are empty, we obtain the so-called block angular structure with linking constraints, where these are defined by the *D* block. Neglecting those linking constraints, a solution could be obtained by solving the (independent) problems defined by each *A* matrix (with a slight abuse of terminology, since the problems are also defined by the senses and right-hand-sides of the constraints and by an objective function not represented). Assuming that the *F* and the *D* matrices are empty, we obtain the so-called block angular structure with linking variables. If none of the *E*, *F*, and *D* matrices are empty matrices a block angular structure with linking constraints and variables is obtained.

Block angular structures with linking variables and with linking variables and constraints are outside the scope of this work. Here we only point out that methods, such as Benders decomposition (Benders, 1962) and cross decomposition (as an example, see (Roy, 1986)) were developed to deal with that kind of structures.

For problems with block angular structure with linking constraints, which are represented in Figure 2.2 for clarity, we will focus on price decomposition (Dantzig-Wolfe / Lagrangean relaxation); other decomposition approaches and methods for those models, such as basis partitioning (Rosen, 1964) and resource directive decompositions (see (Minoux, 1986)) will be briefly presented in Chapter 3 in the context of multicommodity flow problems.

A possible match of this model representation with the examples given above is given in Table 2.2.

It is important to note that different perspectives of a given problem, or different orderings of the variables and constrains of a model, give rise to different structures. This will be discussed in more detail in Section 2.5.2.

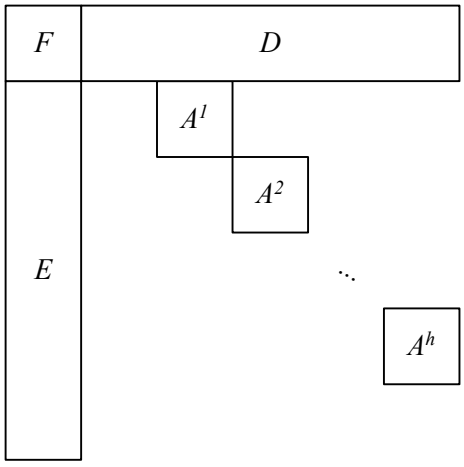



Figure 2.1 Schematic representation of a structured linear/integer programming model.

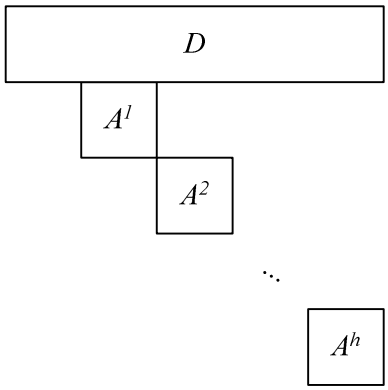



Figure 2.2 Schematic representation of a block angular with linking constraints model.

| Problem | *D* block | *A* blocks |
|---|---|---|
| Production planning | Availability of common resources required for production (for example, capacities of the machines). | One block for each product. Production requirements of each product (for example, forced by existant demand). |
| Vehicle routing | Constraints imposed on the fleet of vehicles (for example, it must visit all the clients). | One for each vehicle. Requirements of the routes and of each vehicle (for example, a route must end at a depot and its capacity cannot be exceeded). |
| Generalised assignment | Constraints imposed on the group of agents (all the tasks must be performed). | One for each agent, related with its capacity. |
| Machine scheduling | Constraints imposed on the jobs (for example, all the jobs must be performed). | One for each machine. Constraints imposed on each machine (for example, two tasks cannot be made at the same time). |

Table 2.2 Examples of models resulting from structured problems.

### 2.2.2 Dantzig-Wolfe decomposition principle

The DWD principle can be applied to any linear programming model. However, its potential is revealed when considering models with the block angular structure with linking constraints described in the previous subsection. We will refer to that case whenever we think it is worth, but, for clearness of notation and exposition, in this Chapter, we will concentrate on the general linear programming model:

$$
\begin{aligned}
Z_{LnP} = \quad & Min\ c\,x && (LnP)\\
& subject\ to: &&\\
& D\,x \geq d && (2.1)\\
& A\,x = b && (2.2)\\
& x \geq 0, &&
\end{aligned}
$$

where $x$ is a column vector of dimension $n$ in which each element, indexed by $j$, is associated with a decision variable, $x_j, j=1,...,n$; $c$ is a row vector with the same dimension: $c = [c_1 \dots c_n]$; $d$ is a column vector with dimension $g$; $D$ is a $g \times n$ matrix; $b$ is a column vector with dimension $m$ and $A$ is a $m \times n$ matrix.

We refer to *(LnP)* as the original formulation and to the decision variables of this model as the original variables. In Section 2.5, we will consider that the decision variables must be integers.

We define the set $S_{SP} = \{ x : Ax = b, x \geq 0 \}$ and the problem

$$Z_{SP} = \quad Min\ \bar{c}\ x \qquad (SP)$$
$$subject\ to:$$
$$x \in S_{SP},$$

where the dimensions of $\bar{c}$ are the same as the ones of $c$.

Every feasible solution of *(SP)* can be represented as a convex combination of the extreme points plus a nonnegative combination of the extreme rays of $S_{SP}$, according to the Minkowski theorem (see, for example, (Nemhauser and Wolsey, 1999)). We define $P$ and $R$ as the sets of indices of all extreme points and extreme rays of $S_{SP}$, respectively; $y^p$ as the $p$-th extreme point and $u^r$ as the $r$-th extreme ray of $S_{SP}$.

Thus, $x \in S_{SP}$ is equivalent to the existence of nonnegative scalars $\lambda_p$, $p \in P$, and $\mu_r$, $r \in R$, associated with the extreme points and extreme rays of $S_{SP}$ such that

$$x = \sum_{p \in P} y^p \lambda_p + \sum_{r \in R} u^r \mu_r \qquad (2.3)$$
$$\sum_{p \in P} \lambda_p = 1$$
$$\lambda_p \geq 0, \ \forall p \in P$$
$$\mu_r \geq 0, \ \forall r \in R.$$

Performing an exchange of variables, model *(LnP)* is equivalent to

$$Z_{LDW} = \quad Min \sum_{p \in P} (c\, y^p)\, \lambda_p + \sum_{r \in R} (c\, u^r) \mu_r \qquad (LDW)$$
$$subject\ to:$$
$$\sum_{p \in P} (D\, y^p) \lambda_p + \sum_{r \in R} (D\, u^r) \mu_r \geq d \qquad (2.4)$$
$$\sum_{p \in P} \lambda_p = 1 \qquad (2.5)$$
$$\lambda_p \geq 0, \ \forall p \in P \qquad (2.6)$$
$$\mu_r \geq 0, \ \forall r \in R. \qquad (2.7)$$

We denote model *(LDW)* as master model and its decision variables, $\lambda_p$ and $\mu_r$, as weight variables.

Constraints *(2.4)* force a feasible solution with respect to the constraints that were not included in the definition of $S_{SP}$. Constraints *(2.5)*, *(2.6)*, and *(2.7)* force a feasible solution of *(LDW)* to belong to $S_{SP}$. Constraints *(2.5)* are referred to as convexity constraints.

A solution expressed in terms of the original variables can always be obtained from a feasible solution of *(LDW)*, both with the same value, by using *(2.3)*.

A feasible solution of *(LnP)* can be mapped into a solution of *(LDW)*, not necessarily in a unique way, since the representation of a point of a set as a convex combination of the extreme points plus a nonnegative combination of the extreme rays of the same set is not unique.

If the original problem is unfeasible or unbounded, the reformulated problem will also be unfeasible or unbounded, respectively.

In general, the master model has a huge number of decision variables: the sum of all extreme points and extreme rays of $S_{SP}$. The enumeration of all those variables is out of question. In fact, only a small number will have a positive value in an optimal solution. On the other side, constraints *(2.2)* were replaced by only one constraint. Column generation is a method to deal with this huge number of variables.

**Example 2.1**

The data of this example is taken from (Bazaraa and Jarvis, 1977).

We consider the linear programming problem:

$$\begin{aligned} &Min\ -x_1 - 2x_2 \\ &subject\ to: \\ &-x_1 - x_2 \geq -12 \\ &x \in S_{SP}, \end{aligned}$$

where

$$S_{SP} = \{\, x: -x_1 + x_2 \leq 2,\ -x_1 + 2x_2 \leq 8,\ x_1 \geq 0,\ x_2 \geq 0 \,\}.$$

In Figure 2.3 a graphical representation of this problem is given; the dotted line represents the constraint that is kept in the master problem. Note that the set $S_{SP}$ is unbounded. Its extreme points are $[0\ 0]^T$, $[0\ 2]^T$, and $[4\ 6]^T$. Its extreme rays are $[1\ 0]^T$ and $[2\ 1]^T$.

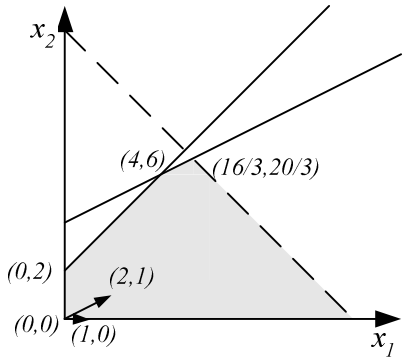



Figure 2.3 Graphical representation of the Example 2.1 problem.

The master problem is:

$$Min\ [-1\ -2]\ [\ 0\ 0\ ]^T\lambda_1 + [-1\ -2]\ [\ 0\ 2\ ]^T\lambda_2 + [-1\ -2]\ [\ 4\ 6\ ]^T\lambda_3 + [-1\ -2]\ [\ 1\ 0\ ]^T\mu_1 + [-1\ -2]\ [\ 2\ 1\ ]^T\mu_2$$

$$subject\ to:$$

$$[-1\ -1]\ [\ 0\ 0\ ]^T\lambda_1 + [-1\ -1]\ [\ 0\ 2\ ]^T\lambda_2 + [-1\ -1]\ [\ 4\ 6\ ]^T\lambda_3 + [-1\ -1]\ [\ 1\ 0\ ]^T\mu_1 + [-1\ -1]\ [\ 2\ 1\ ]^T\mu_2 \geq -12$$

$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

$$\lambda_1,\ \lambda_2,\ \lambda_3,\ \mu_1,\ \mu_2 \geq 0,$$

or, equivalently,

$$Min\ -4\lambda_2 - 16\lambda_3 - \mu_1 - 4\mu_2$$

$$subject\ to:$$

$$-2\lambda_2 - 10\lambda_3 - \mu_1 - 3\mu_2 \geq -12$$

$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

$$\lambda_1,\ \lambda_2,\ \lambda_3,\ \mu_1,\ \mu_2 \geq 0.$$

The optimal solution of the master problem is $\lambda_3=1$, $\mu_2=2/3$, $\lambda_1=\lambda_2=\mu_1=0$, thus the optimal solution of the original problem is $x_1=16/3$, $x_2=20/3$ with value $-56/3$.

♦

Now we consider models with block angular with linking constraints structure. In this case the $A$ matrix is block diagonal and the constraints can be represented as shown in Figure 2.4.

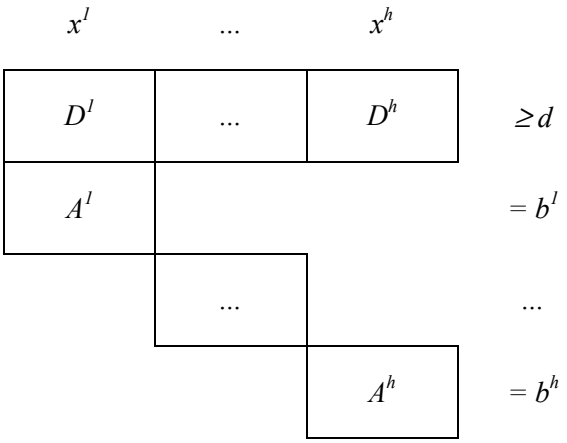



Figure 2.4 Block angular with linking constraints structure.

The original formulation can be rewritten in the following way:

$$Min \sum_{k \in K} c^k x^k$$

*subject to:*

$$\sum_{k \in K} D^k x^k \geq d$$

$$A^k x^k = b^k, \forall k \in K$$

$$x^k \geq 0, \ \forall k \in K,$$

where $K$ is the set of indices of the blocks, $K = \{1,...,h\}$.

Defining a set $S_{SP}{}^k$ for each block, $k \in K$, and representing its feasible solutions through its extreme points and rays, the master problem is:

$$Min \sum_{k \in K} \left( \sum_{p \in P^k} \left(c^k y^{pk}\right) \lambda_{pk} + \sum_{r \in R^k} \left(c^k u^{rk}\right) \mu_{rk} \right)$$

*subject to:*

$$\sum_{k \in K} \left( \sum_{p \in P^k} \left(D^k y^{pk}\right) \lambda_{pk} + \sum_{r \in R^k} \left(D^k u^{rk}\right) \mu_{rk} \right) \geq d$$

$$\sum_{p \in P^k} \lambda_{pk} = 1, \ \forall k \in K$$

$$\lambda_{pk} \geq 0, \ \forall k \in K, \ \forall p \in P^k$$

$$\mu_{rk} \geq 0, \ \forall k \in K, \forall r \in R^k,$$

where $P^k$ and $R^k$ are the sets of indices of all extreme points and extreme rays of the $k$-th block, respectively; $y^{pk}$ is the $p$-th extreme point and $u^{rk}$ is the $r$-th extreme ray of the $k$-th block.

**Example 2.2**

As an example of the application of the DWD principle, we consider a simple production planning problem. We intend to determine the quantities to produce of a set of products in a set of time periods where the demands are known, in order to minimise the total cost composed by holding and production costs. In addition, there is a limit to the quantity of all products that can be produced in each period.

Defining the original variables as the quantity to produce of each product in each time period, denoted by $x_{jk}$, where $j$ is the period index and $k$ is the product index, we obtain the following original model:

$$
\begin{aligned}
&Min \sum_{k=1}^{h}\sum_{j=1}^{n} (c_{jk}x_{jk}+h_{jk}s_{jk}) \\
&\textit{subject to:} \\
&\sum_{k=1}^{h} x_{jk} \le d_j\,,\ j=1,...,n \\
&x_{1k} - s_{1k} = b_{1k}\,,\ k=1,...,h \\
&s_{(j-1)k} + x_{jk} - s_{jk} = b_{jk}\,,\ j=2,...,n,\ k=1,...,h \qquad (2.8) \\
&x_{jk},\ s_{jk} \ge 0,\ j=1,...,n,\ k=1,...,h,
\end{aligned}
$$

where $s_{jk}$ are auxiliary decision variables representing the available stock of product $k$ at the end of period $j$, $n$ is the number of periods, $h$ is the number of products, $c_{jk}$ is the unit production cost of product $k$ in period $j$, $h_{jk}$ is the unit holding cost of product $k$ in period $j$, $d_j$ is the production capacity in period $j$ and $b_{jk}$ is the demand of product $k$ in period $j$.

The original formulation for a three period, two product example is given in Figure 2.5.

If constraints *(2.8)* are used to define the sets $S_{SP}^{k}$ in a DWD, they are associated with production plans (quantities to produce in each period) of product $k$ – defined by the block $k$ constraints. Those sets are bounded, and so do not have extreme rays, and the reformulated model is

$$
\begin{aligned}
&Min \sum_{k\in K}\sum_{p\in P^k} \left(c^k y^{pk}\right)\lambda_{pk} \\
&\textit{subject to:} \\
&\sum_{k\in K}\sum_{p\in P^k} \left(D^k y^{pk}\right)\lambda_{pk} \le d \\
&\sum_{p\in P^k} \lambda_{pk} = 1,\ \forall k\in K \\
&\lambda_{pk} \ge 0,\ \forall k\in K,\ \forall p\in P^k,
\end{aligned}
$$

where $y^{pk}$ denotes the $p$-th production plan of $k$-th product. For the two product, three period

example, we illustrate in Figure 2.6 the reformulated model with two columns associated with two production plans of product 1, namely $y^{11}=[0\ b_{11}\ 0\ b_{21}\ 0\ b_3{}^1]^T$ (producing the required demand in the same period) and $y^{21}=[b_{21}\ \ b_{11}+b_{21}\ 0\ 0\ 0\ b_{31}]^T$ (producing in period 1 satisfying the demands of periods 1 and 2, and producing in period 3 satisfying its demand).

| $s_{11}$ | $x_{11}$ | $s_{21}$ | $x_{21}$ | $s_{31}$ | $x_{31}$ | $s_{12}$ | $x_{12}$ | $s_{22}$ | $x_{22}$ | $s_{32}$ | $x_{32}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | | | | 1 | | | | | $\leq$ | $d_1$ |
| | | | 1 | | | | | | 1 | | | $\leq$ | $d_2$ |
| | | | | | 1 | | | | | | 1 | $\leq$ | $d_3$ |
| –1 | 1 | | | | | | | | | | | = | $b_{11}$ |
| 1 | | –1 | 1 | | | | | | | | | = | $b_{21}$ |
| | | 1 | | –1 | 1 | | | | | | | = | $b_{31}$ |
| | | | | | | –1 | 1 | | | | | = | $b_{11}$ |
| | | | | | | 1 | | –1 | 1 | | | = | $b_{21}$ |
| | | | | | | | | 1 | | –1 | 1 | = | $b_{31}$ |
| $h_{11}$ | $c_{11}$ | $h_{21}$ | $c_{21}$ | $h_{31}$ | $c_{31}$ | $h_{12}$ | $c_{12}$ | $h_{22}$ | $c_{22}$ | $h_{32}$ | $c_{32}$ | | |

Figure 2.5 Simple lot sizing original model for a two product, three period example.

| Variables | | $\lambda_{11}$ | $\lambda_{21}$ | ... | |
|---|---|---|---|---|---|
| Linking constraints | Period 1 | 1 | 1 | ... | $\leq d_1$ |
| | Period 2 | 1 | | ... | $\leq d_2$ |
| | Period 3 | 1 | 1 | ... | $\leq d_3$ |
| Convexity constraints | Product 1 | 1 | 1 | ... | = 1 |
| | Product 2 | | | ... | = 1 |
| Objective function | | $c_{11}b_{11}+c_{21}b_{21}+c_{31}b_{31}$ | $c_{11}(b_{11}+b_{21})+h_{11}b_{21}+c_{31}b_{31}$ | ... | |

Figure 2.6 Part of the simple lot sizing master model for a two product, three period example, resulting from a decomposition by product.

♦

### 2.2.3 Column generation

Column generation is a method used to solve the master problem of the DWD principle. The main idea of this method is that only the extreme points and rays of $S_{SP}$ that are needed to define the optimal basis must be considered in the master problem *(LDW)*. Since, of course, prior to the optimisation the optimal basis is not known, the inclusion of variables is made iteratively, starting from a restricted solution space.

We define two subsets of $P$ and $R$, $\overline{P} \subseteq P$ and $\overline{R} \subseteq R$. A restricted master problem (RMP) is then:

$$Z_{RMP} = \quad Min \sum_{p\in\overline{P}} (c\, y^p)\lambda_p + \sum_{r\in\overline{R}} (c\, u^r)\mu_r \qquad (RMP)$$

*subject to:*

$$\sum_{p\in\overline{P}} (Dy^p)\lambda_p + \sum_{r\in\overline{R}} (Du^r)\mu_r \geq d \qquad (2.9)$$

$$\sum_{p\in\overline{P}} \lambda_p = 1 \qquad (2.10)$$

$$\lambda_p \geq 0, \ \forall p\in\overline{P}$$

$$\mu_r \geq 0, \ \forall r\in\overline{R}\,.$$

We assume that *(RMP)* has at least one feasible solution. In Section 2.4.2 we will discuss the case where finding sets $\overline{P}$ and $\overline{R}$ such that this assumption holds is non-trivial.

According to the linear programming theory, the optimal solution of *(RMP)* is an optimal solution for *(LDW)* if there are no variables outside *(RMP)* with negative reduced cost.

We define the vector $w\geq 0$ and the scalar $\pi$ as the duals associated with constraints *(2.9)* and *(2.10)*, respectively. The reduced cost of one variable $\lambda_p$ is given by $cy^p - wDy^p - \pi$. The reduced cost of one variable $\mu_r$ is given by $cu^r - wDu^r$. At a given dual point, $(\overline{w},\overline{\pi})$, the variable with the most negative reduced cost is the one associated with the optimal solution of the subproblem:

$$Z_{SP\overline{w}} = \quad Min\ (c - \overline{w}\, D)\, x \qquad (SP\,\overline{w})$$

*subject to:*

$$x\in S_{SP},$$

noting that $\overline{\pi}$ is a constant.

If $(SP\,\overline{w})$ is unbounded that means an extreme ray can be detected and the associated $\mu$ variable should be inserted in *(RMP)*. Otherwise, an extreme point is found. If $Z_{SP\overline{w}} < \overline{\pi}$ then the variable associated with the extreme point has a negative reduced cost and should be inserted in *(RMP)*. If there are no variables with negative reduced cost, which means that the optimal solution to *(RMP)* is also an optimal solution to *(LDW)*.

In every iteration of a column generation algorithm, a RMP is solved, providing optimal dual variables; based on the duals, the subproblem is solved and variables are inserted in the RMP, accordingly.

**Example 2.1** (continued from page 17)

We initialise the RMP with the first extreme point (origin) and the first extreme ray. In the original space that corresponds to restricting the feasible set associated with the subproblem constraints to the $x_1$ axis.

The optimal solution of this RMP is $\lambda_1=1$ and $\mu_1=12$. A dual optimal solution is $\overline{w}=1$ and $\overline{\pi}=0$. The subproblem is

$$
\begin{aligned}
&Min\ (-1+w)\, x_1 + (-2+w)\, x_2 \\
&subject\ to: \\
&-x_1 + x_2 \le 2 \\
&-x_1 + 2x_2 \le 8 \\
&x_1, x_2 \ge 0.
\end{aligned}
$$

For $\overline{w}=1$, the subproblem is unbounded along the extreme ray $[2\ 1]^T$. Variable $\mu_2$ is inserted in the RMP, which is (re)optimised. Its optimal dual solution is $\overline{w}=4/3$ and $\overline{\pi}=0$.

For $\overline{w}=4/3$, the optimal solution of the subproblem is the extreme point $[4\ 6]^T$ with value $-8/3$. Since $-8/3<\overline{\pi}$, $\lambda_3$ is inserted in the RMP. Its optimal dual solution is $\overline{w}=4/3$ and $\overline{\pi}=-8/3$.

The optimal solution of the subproblem is the same of the previous iteration. Since $\overline{\pi}=-8/3$ there are no attractive columns and the optimal solution to the RMP is optimal to the overall problem.

♦

### 2.2.4 Linear programming dual and duality gap

In every iteration of the column generation algorithm an upper bound to the value of the optimal solution is obtained by solving the RMP, $Z_{RMP}$.

The linear programming dual of *(LDW)* is:

$$
\begin{aligned}
W_{LDWD} = \quad & Max\ wd + \pi && (LDWD) \\
& subject\ to: && \\
& \pi \le cy^p - wDy^p, \ \forall p \in P && (2.11) \\
& 0 \le cu^r - wDu^r, \ \forall r \in R && \\
& w \ge 0, &&
\end{aligned}
$$

where the (nonnegative) $w$ dual variables are associated with constraints *(2.4)* and (unrestricted in sign) $\pi$ variable is associated with constraint *(2.5)*.

Any feasible solution to *(LDWD)* provides a lower bound to the value of its optimal solution. However, the optimal dual solution that we have, after solving *(RMP)*, is not necessarily feasible for *(LDWD)* since having a primal RMP implies having a relaxed dual master (RDM) problem:

$$
\begin{aligned}
W_{RDM} = \quad & Max\ wd + \pi && (RDM) \\
& subject\ to: &&
\end{aligned}
$$

$$\pi \leq cy^p - wDy^p, \ \forall p \in \overline{P}$$
$$0 \leq cu^r - wDu^r, \ \forall r \in \overline{R}$$
$$w \geq 0.$$

Still, obtaining the value of the dual solution in *(LDWD)* is possible when solving the subproblem if there are no attractive extreme rays (in this case, we consider that the lower bound is $-\infty$). When solving the subproblem at a given point $\overline{w}$, if there are no attractive extreme rays, we are selecting, from all the extreme points, the one that implies the lowest right-hand side in *(2.11)*, as can be seen by the objective function of $(SP\,\overline{w})$. Since $\overline{w}\,d$ is constant, the minimum value of the right-hand sides will imply the maximum possible value of the objective function. Thus, the value of the objective function in *(LDWD)* is given precisely by $\overline{w}\,d + Z_{SP\overline{w}}$. By weak duality, that value is a lower bound to $W_{LDWD} = Z_{LDW}$.

Being so, a duality gap can be easily calculated through

$$Z_{RMP} - (\overline{w}\,d + Z_{SP\overline{w}}) = \overline{w}\,d + \overline{\pi} - (\overline{w}\,d + Z_{SP\overline{w}}) = \overline{\pi} - Z_{SP\overline{w}}.$$

This result allows obtaining optimal solutions with the desired accuracy.

The optimality conditions for linear programming state that a primal-dual pair is optimal if it is primal feasible, dual feasible and respects the complementary slackness conditions. When using column generation we seek a primal-dual pair of solutions that verifies those conditions for the overall problem.

Column generation guarantees the complementary slackness conditions in all iterations when obtaining an optimal solution to the RMP (assuming an optimal extreme point is found as in simplex algorithms). Primal feasibility is assured in the RMP.

Now we turn to dual feasibility, which is checked in the subproblem. The dual solution $(\overline{w}, \overline{\pi})$ may be not feasible because of the constraints that are not present in *(RDM)*. Dual feasibility is achieved by setting $\overline{\pi}' = Z_{SP\overline{w}}$, which amounts to obtaining the extreme point $p \in P$ with the minimum right-hand side in *(2.11)*. If $p \in \overline{P}$ then $\overline{\pi} = \overline{\pi}'$ and the solution is dual feasible already, and thus optimal for the overall problem. If $p \notin \overline{P}$, a dual feasible solution may be obtained by setting $\overline{\pi}' = Z_{SP\overline{w}}$ (which enables the computation of the lower bound previously presented). In that case, by changing the dual solution ($\overline{\pi}$ to $\overline{\pi}'$), complementary slackness conditions are no longer verified, that is, the primal-dual solutions are no longer complementary.

Consider that the dual constraint associated with the first extreme point of the subproblem has no slack: $\overline{\pi} = cy^1 - wDy^1$. By complementary slackness, $\lambda_1 > 0$ (in the absence of primal degeneracy). Since we changed the value of $\overline{\pi}$ to $\overline{\pi}'$ to gain dual feasibility, this constraint has

now a positive slack, thus violating $(\bar{\pi}' - cy^l + wDy^l)\lambda_l = 0$. A new iteration begins with the computation of complementary primal-dual solutions that minimise-maximise RMP-RDM, with the inclusion of a new variable-constraint.

Summarising, in every iteration of the column generation algorithm primal feasibility and complementary slackness are ensured when solving the RMP. Dual feasibility is checked in the subproblem. When checking dual feasibility it is possible to find a dual feasible solution that allows computing a lower bound to $Z_{LDW}$.

### 2.2.5 Columns removal and convergence

Column generation can be viewed as an extension of the primal simplex algorithm. In all iterations, a basic solution is determined by solving the current RMP and the most promising nonbasic variable is determined by solving the subproblem. However, since the number of columns of the RMPs can become quite large (given their exponential size), strategies for columns removal may have a particular importance when implementing a column generation based algorithm.

Two extreme situations can be considered when defining a strategy for columns removal: never deleting variables of the RMPs or deleting all nonbasic columns of the current RMP in every iteration.

In the first situation, finite convergence is guaranteed, as long the simplex implementation employed to solve the RMPs uses techniques for dealing with cycling (which is usually the case). Cycling can occur if a (primal) basis is degenerate, meaning that there are basic variables with zero value (see, for example, (Murty, 1983)). In the presence of a degenerate basis, if the choice of the leaving (degenerate basic) variable and entering variable (one with non-positive reduced cost) is arbitrary, a cycle of basis, which does not include the one that allows the strict improvement of the solution value (or which allows the detection that the optimal solution was found), can be formed. Note that for a variable to play a part in that cycle of basis, it must have a negative reduced cost at some iteration, thus, it will be generated by the subproblem. As long as variables are not deleted, cycle prevention is then a task for the RMP solver and does not pose any difficulties for the implementation of a column generation algorithm.

In the second situation, finite convergence is not theoretically guaranteed. When all nonbasic variables are removed from the RMP, the subproblem may suggest a variable to enter the basis that will lead to a basis already considered in a previous iteration. Note that, since the variables outside the RMP are being implicitly considered, it is not simple to apply any lexicographical rules to select the entering variable, which would be a possibility to deal with cycling.

Although columns removal may compromise the convergence of column generation, in practise, implementations where columns removal is performed are frequent. We will discuss some of those strategies in subsection 2.4.2.

## 2.3 Dantzig-Wolfe Decomposition and Lagrangean Relaxation

### 2.3.1 Lagrangean relaxation

In the previous Section, we took a linear programming perspective over the DWD principle and column generation. A different perspective can be taken if we consider its relation with Lagrangean relaxation. Although linear programming duality and Lagrangean relaxation applied in linear programming can be seen as the same thing, we think this different perspective is worth, given the own importance of Lagrangean relaxation. Furthermore, this perspective allows contextualising column generation within the methods for non-differentiable optimisation.

The use of Lagrange multipliers for obtaining solutions of optimisation problems dates back to the XVIII century, when Joseph Louis Lagrange (1736-1813) lived. Their use in nonlinear programming has accompanied that discipline from its origins, in the middle XX century, to the present.

The term "Lagrangean relaxation" was coined in the middle 1970s by A. M. Geoffrion (Geoffrion, 1974) in the context of obtaining lower bounds in integer programming. This application of Lagrangean relaxation became relevant with the work of M. Held and R. M. Karp on the travelling salesman problem (Held and Karp, 1970; Held and Karp, 1971) in the beginning of the same decade, where the subgradient method was first used in that context. Surveys about Lagrangean relaxation can be found in (Shapiro, 1979; Fisher, 1981; Bazaraa et al., 1993; Beasley, 1995; Bertsekas, 1999).

In Lagrangean relaxation, the minimisation original problem *(LnP)* (defined in page 15) is replaced by a closely related maximisation problem:

$$\begin{aligned} Z_{LgP} = \quad & Max\ \varphi(w), && (LgP)\\ & subject\ to:\\ & w \geq 0, \end{aligned}$$

where

$$\begin{aligned} \varphi(w) = \quad & Min\ c\,x + w\,(d - Dx) && (LgSP)\\ & subject\ to:\\ & A\,x = b \end{aligned}$$

$$x \geq 0.$$

Problem *(LgP)* is the Lagrangean dual problem, $\varphi(w)$ the Lagrangean dual function and *(LgSP)* the Lagrangean subproblem. The Lagrangean subproblem is obtained by associating a vector of dimension $g$ of nonnegative dual variables (Lagrange multipliers), $w \geq 0$, with constraints *(2.1)*, which, in this way, are penalised in the objective function if they are not satisfied.

Reminding that we are assuming that the original problem is a linear programming one, two fundamental results are:

$$Z_{LgP} = Z_{LnP};$$
$$\varphi(w) \leq Z_{LnP}\ , \forall w \geq 0.$$

The evaluation of the Lagrangean dual function at a point $\overline{w}$ is made by solving *(LgSP)* at that point. If the *(LgSP)* is unbounded that means there exists an extreme ray, $u^r$, such that $cu^r - wDu^r < 0$ (noting that $\overline{w}\, d$ is a constant). In that case, $\varphi(\overline{w}) = -\infty$, since it was proved that it is not that point that maximises $\varphi(w)$. Points that may be optimal of *(LgP)* must satisfy the constraints $cu^r - wDu^r \geq 0,\ \forall r \in R$.

If, at $\overline{w}$, there exists a finite optimal solution, there is at least one extreme point of $S_{SP}$ that is an optimal solution of *(LgSP)*, thus $\varphi(\overline{w}) = \underset{p \in P}{Min} \{ cy^p + \overline{w}\ (d - Dy^p) \}$, where $P$ denotes the index set of the extreme points of $S_{SP}$. Noting that $cy^p + w\ (d - Dy^p),\ \forall p \in P$, are linear functions, $\varphi(w)$ is a concave piecewise linear function with breakpoints where *(LgSP)* has alternative optimal solutions.

**Example 2.1** (continued from page 17)

Associating a Lagrange multiplier with the constraint $-x_1 - x_2 \geq -12$, we obtain the Lagrangean dual function showed in Figure 2.7. The two vertical dotted lines are associated with the extreme rays of *(LgSP)*, points to their left have $\varphi(w) = -\infty$. The other three dotted lines represent the linear functions associated with the extreme points of *(LgSP)*.

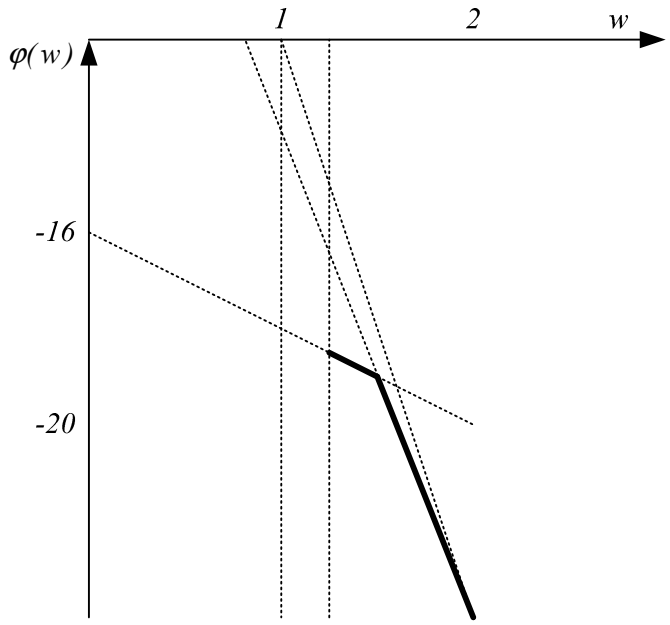



Figure 2.7 Lagrangean dual function of the Example 2.1.

♦

### 2.3.2 Equivalence between Lagrangean relaxation and Dantzig-Wolfe decomposition

The equivalence between DWD and Lagrangean relaxation is formally proven in, for example, (Nemhauser and Wolsey, 1999) and is a consequence of the equivalence of dualisation and convexification (Magnanti et al., 1976).

The value of the Lagrangean function at a point $\overline{w}$, where $\varphi(\overline{w})$ is finite, is given by $\underset{p\in P}{Min}\ \{ cy^p + \overline{w}\ (d - Dy^p\ \}$. Being so, the Lagrangean dual problem can be solved by linear programming:

$$\begin{aligned}
&Max\ \phi && (LgLnP)\\
&subject\ to:\\
&\phi \le cy^p + w(d - Dy^p),\ \forall p \in P && (2.12)\\
&0 \le cu^r - wDu^r,\ \forall r \in R && (2.13)\\
&w \ge 0,
\end{aligned}$$

where constraints *(2.13)* exclude points where $\varphi(\overline{w}) = -\infty$ and constraints *(2.12)* force the value of the Lagrangean dual function to be defined by the optimal value of *(LgSP)*.

**Example 2.1** (continued from page 27)

The Lagrangean dual problem can be solved by the following linear programming problem.

$$
\begin{aligned}
&Max\ \phi \\
&subject\ to: \\
&\phi \le [-1\ -2]\ [0\ 0]^T + w\ (-12 - [-1\ -1][0\ 0]^T) \\
&\phi \le [-1\ -2]\ [0\ 2]^T + w\ (-12 - [-1\ -1][0\ 2]^T) \\
&\phi \le [-1\ -2]\ [4\ 6]^T + w\ (-12 - [-1\ -1][4\ 6]^T) \\
&0 \le [-1\ -2]\ [1\ 0]^T + w\ [-1\ -1][1\ 0]^T \\
&0 \le [-1\ -2]\ [2\ 1]^T + w\ [-1\ -1][2\ 1]^T \\
&w \ge 0.
\end{aligned}
$$

The optimal solution is given by $w=4/3$ and $\phi=-56/3$.

♦

At a point $\overline{w}$, given that $\overline{w}\,d$ is a constant, $\varphi(\overline{w}) = \overline{w}\,d + \underset{p\in P}{Min}\ \{\ cy^p - \overline{w}\,Dy^p\ \}$, thus problem *(LgLnP)* is equivalent to *(LDWD)* (defined in page 23), making $\phi = \pi + wd$.

Primal-dual equivalence between DWD and Lagrangean relaxation translates into the primal-dual equivalence between column generation and the Kelley's cutting plane method (KCPM). In that cutting plane method, a relaxed master dual is considered and, in each iteration, the violation of the constraints that are not present in *(RDM)* is checked by solving the subproblem at the current point given by the optimal solution of *(RDM)*.

### 2.3.3 Optimality conditions and primal solutions

A dual feasible point, $w^*$, is an optimal solution to the Lagrangean dual problem if and only if there is an optimal solution of *(LgSP)*, $y^*$, at $w^*$ such that:

*(i)* $Dy^* \ge d$ (primal feasibility);

*(ii)* $w^*(d - Dy^*) = 0$ (complementary slackness).

Furthermore, under these conditions, $y^*$ is an optimal solution to the original problem.

It is important to note that it may not be trivial, at an optimal dual point $w^*$, to find a $y^*$ that satisfies *(i)* and *(ii)*. This difficulty lies in the fact that, if *(LgSP)* has alternative optimal solutions at $w^*$, there is no imediate way for the subproblem to select one solution such that conditions *(i)* and *(ii)* are satisfied. An illustration of this is given in Figure 2.8. Points $y^1$, $y^2$, $y^3$, and $y^4$ are the extreme points of the subproblem. The dotted line represents the dualised constraint $D_1x \ge d_1$. Consider that the unique optimal solution of the original problem is $y^*$ and that $w^*$ maximises $\varphi(w)$. By solving the subproblem at $w^*$, all points that form the convex

combination of $y^1$ and $y^2$ are optimal. Note that $y^1$ does not satisfy the complementary slackness condition ($w^* > 0$ is implied by the uniqueness of the optimal solution and $D_1 y^1 > d_1$) and $y^2$ is not primal feasible. The subproblem cannot identify the optimal solution that satisfies *(i)* and *(ii)*.

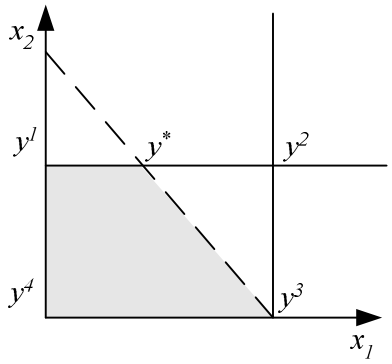



Figure 2.8 Illustration of the difficulty of getting a primal optimal solution based on an optimal solution of the Lagrangean dual function.

Furthermore, the existence of alternative optimal solutions of the subproblem at an optimal dual Lagrangean point, $w^*$, can be seen as natural, since that happens in the breakpoints of the Lagrangean dual function, as illustrated in Figure 2.9, $\varphi(w^*) = cy^1 + w^*(b - Ay^1) = cy^2 + w^*(b - Ay^2)$.

This discussion leads to the fact that, even assuming that an optimal dual solution is known, obtaining a primal optimal solution amounts to determining the best primal feasible convex combination of the extreme points of the subproblem (in order to satisfy optimality conditions *(i)* and *(ii)*). This is precisely what is done when solving the (restricted) master (primal) problem of column generation method (or, equivalently, the relaxed – since not all constraints are being considered – master dual problem of the cutting plane method).

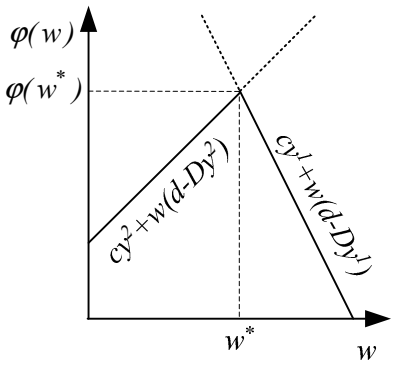



Figure 2.9 Illustration of an optimal breakpoint of the dual Lagrangean function.

### 2.3.4 Methods for solving the Lagrangean dual

#### General considerations

In this subsection, we refer to methods for solving the Lagrangean dual problem *(LgP)*. We briefly review and give references on the subgradient, volume, bundle and analytic center cutting plane methods. These methods deserve particular emphasis for their generalised/promising practical use and/or relation with KCPM / column generation. A description of other methods can be found in (Bertsekas, 1999) and (Goffin and Vial, 1999). Our goal with this subsection is to contextualise the KCPM / column generation method, and thus we do not provide a formal description of the methods, neither discuss their several variants.

Although we focus on solving the Lagrangean dual problem, it is relevant noting that all the methods mentioned here may be applied in any convex/concave programming problem (see (Hiriart-Urruty and Lemaréchal, 1993a; Hiriart-Urruty and Lemaréchal, 1993b) for a much deeper presentation in that broader context).

The Lagrangean dual problem, *(LgP)*, amounts to maximising a concave piecewise linear function, $\varphi(w)$, such as the one represented in Figure 2.10.

As noted earlier, each linear function defining the Lagrangean function is associated with an extreme point of *(LgSP)*, thus their number is exponential, which precludes the possibility of explicitly considering all of them. However, by solving *(LgSP)* at a given point, $\overline{w}$, there are two kinds of useful information that can be obtained: the value of the Lagrangean dual function at $\overline{w}$, $\varphi(\overline{w})$, and a subgradient of $\varphi(w)$ at $\overline{w}$, that is, a vector $s = [s_1 \dots s_m]^T$ such that $\varphi(w) \leq \varphi(\overline{w}) + s\,(w - \overline{w}),\ \forall w$, given by $d - Dy'$, where $y'$ is an optimal solution of *(LgSP)* at $\overline{w}$. That information is sufficient to describe the linear function associated with the extreme point of *(LgSP)* found. (If, at $\overline{w}$, *(LgSP)* does not have a finite solution, then a constraint excluding that point is obtained).

In the methods discussed here there is no control over the subgradient returned by the subproblem: in the presence of alternative optimal solutions, the subproblem returns an arbitrary one. The possibility of obtaining different optimal solutions of the subproblem (in the limit, all of them, that is, obtaining all the subgradients at the point – the subdifferential) is excluded.

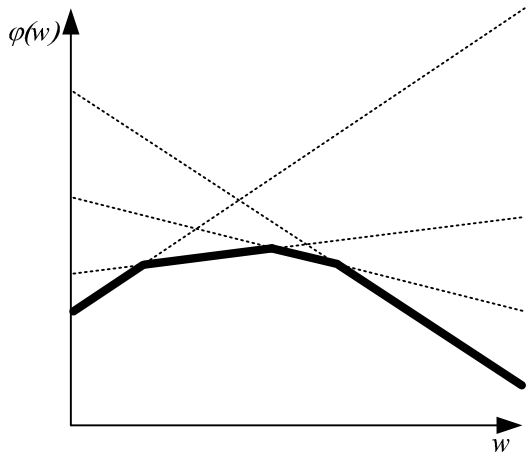



Figure 2.10 Illustration of a Lagrangean dual function.

A generic iteration of a method for solving the Lagrangean dual is as follows.

1. Obtain a trial point $\overline{w}$, based on the information gathered so far.

2. Solve the subproblem *(LgSP)* at $\overline{w}$, obtaining $\varphi(\overline{w})$ and a subgradient of the Lagrangean function at $\overline{w}$ given by $s_{\overline{w}} = d - Dy^p$ where $y^p$ is an optimal solution of *(LgSP)* at $\overline{w}$.

3. Update the available information.

4. If the trial point is accepted move to $\overline{w}$.

5. If the stopping criterion is satisfied, stop. Else go to 1.

**Kelley's cutting plane**

In KCPM, the information used to obtain a trial point is a function that approximates the Lagrangean function, obtained through the points, values, and subgradients of previous iterations. The trial point is a maximiser of that function, obtained by solving a linear programming problem – the RDM. The trial point is always accepted. Since the optimal value of the current RDM gives an upper bound and $\varphi(\overline{w})$ gives a lower bound to the optimal value of the Lagrangean function, the stopping criterion is based on an optimality gap, allowing to obtain primal solutions with the desired accuracy.

The initialisation of this method may require one artificial upper bound, since the first RDMs problems can be unbounded.

**Subgradient**

The KCPM gathers all the information from previous iterations in order to decide the next point to evaluate. An opposite approach, in the sense that very few past detailed information is used in the current iteration, is given by the subgradient method (with roots in the works of Shor

and Poljak in the 1960s, see references in (Held et al., 1974)). The trial point obtained in step 4 is also always accepted, but, in the simpler version of the method, it is only based on the subgradient of the current iteration and on a step size. Convergence is assured taking into account that a sufficiently small step given in the direction of a subgradient (that is not necessarily an ascent direction) results in a point closer to the optimum. In practise, the rules used for determining the step sizes do not, theoretically, assure convergence (see, for example, (Minoux, 1986; Bertsekas, 1999; Nemhauser and Wolsey, 1999)). The main advantage of this method is its simplicity: no special procedure for its initialisation is needed, memory computational requirements are negligible and its implementation is easy. The drawbacks are the lack of a primal perspective and the stopping criterion. Theoretically, a necessary and sufficient condition for the optimality of a point is the existence of a null subgradient at that point. In practise, since the subgradients of previous iterations are discarded (or, in a more elaborated version, aggregated with different weights) the verification of that condition relies exclusively in the subproblem for which, as discussed before, there is no control over the subgradient it returns (as opposed to the KCPM where the master problem is able to generate a point where there is a null subgradient). Thus, in practise, the stopping criterion is usually related with the number of iterations performed or the number of iterations without improvement in the value of the Lagrangean function.

**Volume**

The most diffused method to solve the Lagrangean dual is, undoubtedly, the subgradient method. Besides the fact that it was the first one used in this context, it has, at least, two main advantages: it is easy to code and has minimal memory requirements. However, since it does not keep in memory the solutions of the subproblems solved, it does not guarantee anything (not even feasibility) about the solution of the original problem (the solution of the last subproblem solved).

The volume method (Barahona and Anbil, 2000) can be seen as an extension of the subgradient method designed to provide a primal solution, but still keeping its simplicity. A primal solution, $\bar{x}$, obtained by a convex combination of the primal solutions obtained in previous iterations, is considered in each iteration (the weights are adjusted through the execution of the algorithm). In each iteration a trial point is obtained, based on a step size and on the direction defined by $d - D\bar{x}$. That trial point can be accepted or not as the current point for the next iteration. Three types of iterations can occur. In a red iteration the trial point is not accepted because there has been no improvement. In a green or yellow iteration there is an improvement of the Lagrangean function value and the trial point is accepted. The difference between those types of iterations lies in the way a constant in the step size update expression is modified for the following iteration. In a green iteration the angle between the direction and the

subgradient of the current point is acute, and thus that constant is set to a value larger than the current one. If the angle is not acute, then a yellow iteration is performed, where the constant of the step size expression is reduced. Taking into account that primal feasibility can be slightly violated, but given the existence of a primal solution, the stopping criterion can be based on the gap.

Convergence properties and their relation with bundle methods are discussed in (Bahiense et al., 2002).

The advantages of using subgradient and volume methods instead of the KCPM are clearly application dependent. They are coded more easily and tend to execute faster, but are not intended to give primal solutions with the same quality as the ones given by KCPM.

**Bundle**

Bundle methods have their roots in the work of Lemaréchal in the 1970s (for references, see (Lemaréchal, 1989)). They can be seen as an extension of the KCPM, since, in each iteration, the same function that approximates the Lagrangean function, based on the points and subgradients of previous iterations, is considered. However, the trial point is obtained by solving a quadratic master problem, where points distant from the current one are penalised. In addition, contrary to the KCPM, the trial point is only accepted (serious step as opposed to a null step) as the current one for the next iteration if the predicted improvement given by the approximation is sufficiently close to the real improvement measured when the Lagrangean function is evaluated at the trial point. In a null step, although the current point remains the same, the approximation function is enriched.

When related to the subgradient method, in each iteration of a bundle method, a step size is also adjusted and a direction is also determined. That direction is a convex combination of the subgradients found in previous iterations, and is obtained through the solution of the quadratic master problem, where a measure (linearisation error) of the validity of each subgradient in the current point is taken into account. This allows obtaining a direction where subgradients in points near the current one tend to have a larger weight than subgradients in points far from the current one.

As opposed to the KCPM, bundle methods do not need to use an artificial upper bound in the first iterations since the master problem of the bundle method always has a finite solution. In addition, convergence properties are better because local information is taken into account when deciding the trial point at each iteration. The price to pay is having a much more difficult master problem to solve in each iteration (that is, a nonlinear one), a weaker stopping criterion and primal solutions that can (slightly) violate the original constraints.

In (Briant et al., 2004) computational tests for comparing the bundle and KCPM approaches are given for several instances of different problems. For two versions of a cutting

stock problem (minimising the number of rolls and minimising the total waste) KCPM took less iterations and less time. For a vertex coloring problem, bundle took considerable less iterations but the solution times were similar. For a capacitated vehicle routing, bundle again took considerable less iterations, but much larger solution times. For the travelling salesman problem, the number of iterations of bundle was again much smaller, but time results were not presented. Finally, for a multi-item lot sizing problem, KCPM took much less iterations and solution time.

Bundle requires the use of several parameters that need to be calibrated for the instance/problem at hand. Although there is some evidence that the method is robust (with respect to those parameters), in particular compared with the subgradient method (Crainic et al., 2001), the absence of parameters that need to be calibrated of KCPM may be considered as another advantage of this latter method.

For an in depth treatment of bundle methods and variants see (Medhi, 1994; Lemaréchal et al., 1995; Frangioni, 1997; Frangioni, 2002).

**Analytic center cutting plane**

The analytic center cutting plane method (ACCPM) (Goffin et al., 1992; Goffin et al., 1993) is closely related to the KCPM. The main conceptual difference is that the current point of the next iteration is given by the analytic center of the localization set defined by

$$\pi \geq z_{LB}$$
$$\pi \leq cy^p - w(d - Dy^p)\ ,\ \forall p \in \overline{P}$$
$$0 \leq cu^r - wDu^r,\ \forall r \in \overline{R}\ ,$$

where $z_{LB}$ is the best value of the Lagrangean function found so far, being the remaining notation as introduced before. Using concepts from interior point methods, the analytic center (or an approximation of it) can be efficiently computed in every iteration.

The potential advantage of this approach lies in the fact that a central point contains more information about the Lagrangean function than a maximiser, since it is defined by all the cutting planes generated so far.

Its similarities with KCPM include the possible requirement of one artificial upper bound, the stopping criterion, and the possibility of obtaining optimal primal solutions with the desired accuracy. Still comparing with the KCPM, besides its clear conceptual difference, we must note that an implementation of ACCPM involves a much harder coding effort, although some tools have been developed to make it easier (Péton and Vial, 2001). For references and detailed treatment of the ACCPM see (Goffin and Vial, 1999).

## 2.4 Column Generation Variants

### 2.4.1 Head-in, tail-off, and instability

Theoretically, column generation algorithms have poor convergence properties (Wolfe, 1970; Lemaréchal, 2003). In practise, three phenomena are observed (in some applications), usually denoted by head-in, tail-off, and instability.

A graphical representation of those behaviours of a column generation algorithm is given in Figure 2.11.

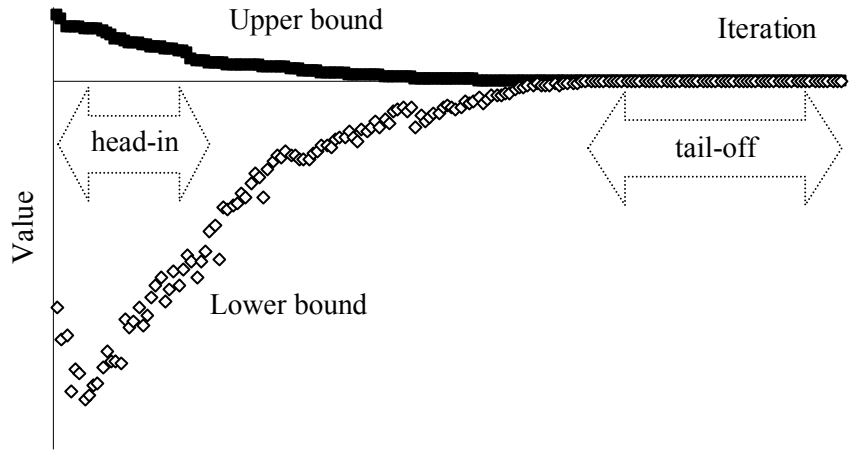



Figure 2.11 Illustration of the head-in and tail-off effects in column generation / KCMP.

The head-in effect is a consequence of the poor quality of the primal and dual information obtained in the first iterations. Particularly when it is not easy to obtain a first RMP that has a feasible solution to the original problem, we may expect that, in each of those iterations, a large number of columns are attractive and the selection of the ones to be inserted in the RMP is more or less arbitrarily, since the dual information that an "artificial" RMP gives is necessarily poor.

In the last iterations, the duality gap may be small, but closing it may be take several iterations, a phenomenon that is known as the tail-off effect. We may consider two reasons for this. Firstly, when the RMP is degenerate, there are alternative dual optimal solutions and the subproblem may have to generate several columns (several iterations) to strictly improve the RMP primal solution. Secondly, as discussed in subsection 2.3.3, page 29, it is natural that the subproblem has alternative optimal solutions, and so the one chosen arbitrarily may not be the one that reduces the gap as intended.

Instability of the column generation refers to the fact that dual variables can take very

different values from one iteration to the next. In Figure 2.12 an illustration is given: the value of the dual variable in the first iteration is closer to its value in the third iteration than to its value in the second iteration. This may lead to wasting iterations approximating the Lagrangean dual function in points far from the optimal, gathering useless cutting planes and lower bounds.

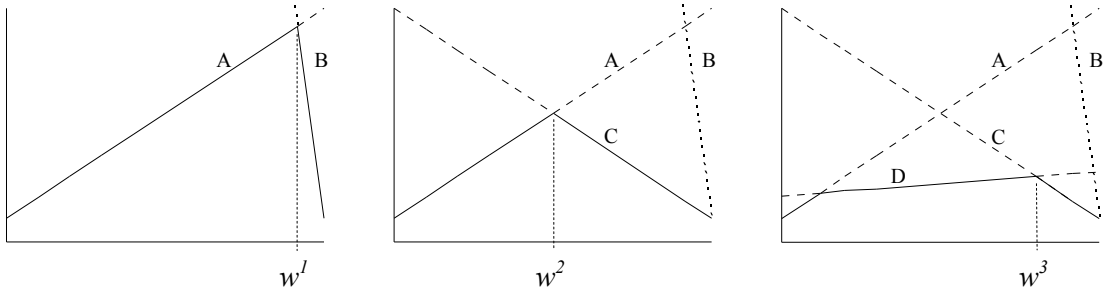



Figure 2.12 Illustration of the instable behaviour of the dual solution in the KCPM.

This oscillatory behaviour of the values of the dual variables in the course of the algorithm leads to significant differences among the lower bounds obtained in consecutive iterations, as illustrated before in Figure 2.11.

Bundle and ACCPM methods may be seen as stabilised versions of column generation / KCPM, dealing with the pernicious behaviours described in the previous paragraphs at the price of not using directly (in the case of the ACCPM, or at all, in the case of bundle methods) standard linear programming techniques.

Clearly, much more comparative results (besides the ones pointed out in the previous subsection) between the several methods must be obtained (and in a larger number of problems), before having a clear picture of how these issues affect their comparative overall efficiency.

In the next two subsections, we detail some approaches developed to alleviate the pernicious behaviours identified above. In the next subsection, we discuss implementation alternatives for column generation. In the last subsection, we focus on stabilisation procedures that keep the main distinctive feature of column generation / KCPM, that is, in each iteration a primal-dual solution that maximises-minimises the current linear programming RMP-RDM is obtained.

### 2.4.2 Column generation implementation variants

**First RMP**

The construction of the first RMP may have an important impact in the head-in effect mentioned above. A column generation algorithm requires the construction of a first RMP,

which must be feasible so that the algorithm may proceed. Clearly, in general, obtaining a set of columns that assure feasibility is not an easy task. The use of artificial variables may be required, implying a phase I, where the objective is to find a feasible solution to the RMP. The classical artificial variables methods, two phase and big M, can be used for that purpose.

The big M method is based on having the artificial variables with a sufficiently large coefficient in the objective function, implying a null value for those variables in a feasible solution for the master problem. A very large value can lead to scaling difficulties of the master solver. Thus, the value of M should be as small as possible. Setting a small value of M has another advantage. Columns that will be generated in this phase I will tend to have a better quality in the sense that their selection has a closer relation with their original cost. If the value of M is very large, all columns suggested by the subproblem are attractive.

The two-phase method avoids scaling difficulties, since the coefficients of the artificial variables are small. Its disadvantage is that the selection of columns does not take into account the original costs.

Artificial variables can be inserted in the linking constraints or in the convexity constraints (or in both). Their judicious use, along with a judicious choice of how the first columns are generated may have an important impact on the efficiency of the column generation algorithm.

Usual approaches for generating that first set of colums are solving exactly the subproblems with the original costs, or solving them heuristically, taking into account the linking constraints (examples of different strategies for two multicommodity flow problems are given in Chapter 3 and Chapter 4).

**RMP**

In each iteration of the column generation algorithm, a RMP is solved. Taking the dual perspective, solving the RMP consists in finding a dual solution that is tested for feasibility (in the overall problem) when solving the subproblem. Column generation algorithms select a dual solution that maximises the Lagrangean function (in opposition to the methods discussed in 2.3.4).

We propose a different alternative that consists in selecting a feasible dual solution not necessarily optimal. In practical terms, this alternative amounts to stopping the RMP optimisation as soon as a dual feasible solution is obtained or a predetermined number of iterations was made (given that the current dual solution is feasible). In some iterations, the RMP is solved exactly to assure convergence.

The rationale behind this alternative is that obtaining the dual optimal solution of the RMP can be too costly and, anyway, that solution can be very far from an optimal solution of the overall problem. In addition, we may expect to get a feasible dual solution closer to the

previous one, something that may attenuate the instability of the dual solutions mentioned above.

Furthermore, this approach may be useful when it is not trivial to find a subset of columns to build a (primal) feasible RMP. In that case, in the first iterations where artificial variables are being used the time consumed solving the RMP optimally may be used with advantage in generating a larger set of columns that, probably, will lead to a feasible primal solution more quickly, alleviating the head-in effect.

**Subproblem**

In a given iteration, all that is needed to improve the current (RMP) solution is one column with a negative reduced cost (neglecting degeneracy), and not necessarily the one with lower reduced cost. This way, the subproblem can generate columns that are not associated with optimal solutions of the subproblem. In addition, several (attractive) columns of the same subproblem can be inserted in the same iteration. Those can be generated with heuristics. Of course, in this case, if no attractive columns are detected heuristically, the subproblem must be solved exactly to prove (or not) the optimality of the current solution.

Another alternative is to obtain the second best solution of the subproblem, besides the optimal solution, and so on (in the limit all the attractive extreme points). Clearly, this depends on the specific subproblem: if the second best solution can be obtained with little computational effort, this approach may be appealing. The rationale behind this is that a larger portion of the dual space is cut or a closer approximation of the Lagrangean dual function is obtained.

**RMP rows and columns management**

In every iteration of the column generation algorithm the number of columns of the RMP increases, which may successively reduce the ability of the linear programming solver to obtain an optimal solution efficiently.

Although, theoretically, as noted in subsection 2.2.5, removing columns from the RMP may affect the finite convergence of the algorithm, in practise, it is quite common to implement strategies for columns removal (in our experience, even with aggressive strategies, such as removing all nonbasic columns in every iteration, convergence was always achieved).

A first alternative is to remove colums with reduced cost greater than a given threshold. A second one is to remove columns with zero value for a predefined number of iterations. There are two other alternatives: to remove columns with reduced cost greater than the current duality gap and to remove columns in such a way that the number of columns of the RMP is limited to a given number. In this last case, the maximum size of a RMP to be optimised by the linear programming solver can be previously defined. The maximum number of columns must be greater than the number of rows of the RMP (which ensures that the number of columns of each

RMP is sufficiently large to form a basis). The selection of the columns to remove may take into account the ones with a smaller probability to be generated again, by sorting them by decreasing reduced costs.

We now turn to the dynamic management of rows, meaning that, in order to solve the RMPs easily, addition and removal of rows may also be performed.

Not taking into account all the rows of the master model means that a relaxed problem is being considered. Of course, if an optimal solution to a relaxed RMP is obtained, it is necessary to check if the constraints that are not present in the RMP are being violated, and, if there are any, to insert them in the RMP.

In the most basic version of dynamic insertion of rows, this procedure can be seen as composed by two cycles. In the inner cycle, a relaxed problem, in which only some rows are considered, is solved by column generation. In the outer cycle, violated rows are detected and the relaxation of the problem is tightened by their insertion in the RMP.

In Figure 2.13, an illustration of the evolution of the upper and lower bounds obtained during this type of process is given. When violated rows are inserted in the RMP, the upper bound increases. Column generation is then used to close the duality gap and the procedure is repeated until there are neither attractive columns nor violated rows.

As a particular example of how this strategy can improve the efficiency of the column generation approach, we note that the instance used to construct Figure 2.13 is the same as the one used for Figure 2.11 (instance *bs01* of the binary multicommodity flow problem addressed in Chapter 4). Without dynamic insertion of rows this instance took 270 seconds and 176 iterations (RMP optimisations) to be solved. With dynamic insertion of rows, it took only 3 seconds and 102 iterations.

Rows removal is another alternative to keep the size of RMP as small as possible. A possible strategy is to remove rows that are inactive for more than a predefined number of iterations. Of course, that parameter should not be set to a very small value, since removed rows may be generated again in a subsequent iteration. As it happens with columns, using aggressive removal strategies allows to keep the RMPs smaller and easier to solve, but it is worth to emphasise that it also may lead to instability in the column generation procedure.

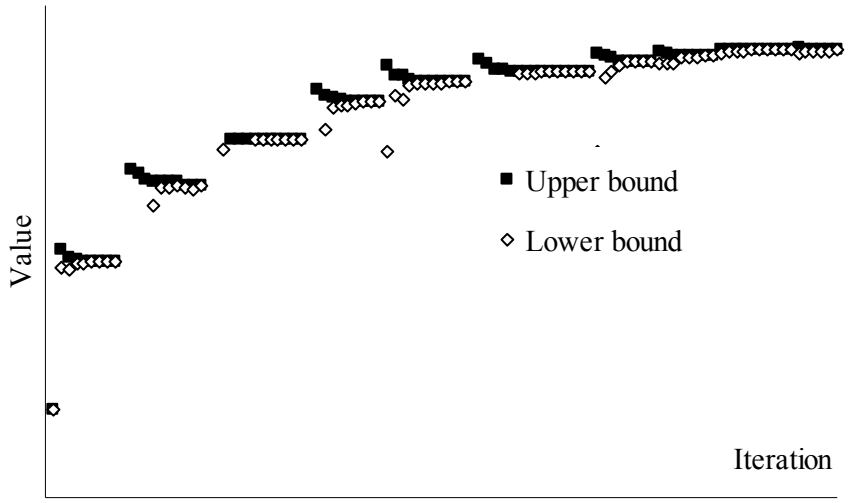



Figure 2.13 Illustration of the effect of dynamic management of rows.

### 2.4.3 Stabilisation

Several procedures have been devised to stabilise the KCPM. One approach is, when solving the RMP, to penalise the dual solutions far from the current one as in bundle methods, but using a penalisation scheme that maintains the RMP a linear problem.

The first described approach of this type is the box step method (Marsten et al., 1975), where lower and upper bounds are associated with each dual variable. Here we denote the RMP of a given iteration by *(P)*, its dual by *(D)*, and represent them as following:

| | | | |
|---|---|---|---|
| $Min\ c\ x$ | *(P)* | $Max\ b\ w$ | *(D)* |
| *subject to:* | | *subject to:* | |
| $A\ x = b$ | | $w\ A \leq c.$ | |
| $x \geq 0,$ | | | |

Forcing the dual variables to lie inside a box amounts to adding lower and upper bounds constraints in *(D)* and variables, represented by the vectors $y$, in *(P)*:

| | | | |
|---|---|---|---|
| $Min\ c\ x - \delta^- y^- + \delta^+ y^+$ | *(P')* | $Max\ b\ w$ | *(D')* |
| *subject to:* | | *subject to:* | |
| $A\ x - y^- + y^+ = b$ | | $w\ A \leq c$ | |
| $x, y^-, y^+ \geq 0,$ | | $\delta^- \leq w \leq \delta^+,$ | |

where $\delta^-$ and $\delta^+$ are vectors of parameters with appropriate dimensions.

In each iteration the box is "placed" around the current dual solution. If the optimal dual

solution is strictly inside the box, then it is an optimal solution to the original problem *(D)*. (Note that, by complementary slackness, the primal variables $y$ are zero). If any dual variable lies in the boundary of the box (its value equals the lower or the upper bound) the box is recentred for the next iteration.

Extensions of this basic approach take into account dynamic updating of the size of the box, possibly depending on the improvement of the lower bound (as in (Kallehauge et al., 2001)). In (Merle et al., 1999) the penalisation scheme is smoothed by allowing dual solutions outside the box, with a linear penalisation. In addition, in the same reference, different dynamic strategies for updating all the parameters involved and different stopping criteria are developed and empirically tested.

A different stabilisation approach is presented in (Wentges, 1997). The fundamental idea is to obtain the dual solution by a convex combination of the one given by the RMP-RDM and the best one found so far (the one that gave the best lower bound), where the weights used are iteration dependent (the weight of the best solution increases in order to assure convergence, as proved in the mentioned reference). Computational results for the capacitated facility location problem confirm the potential of that approach.

Another stabilisation approach is proposed in (Carvalho, 2000). The fundamental idea is to include a set of extra variables (extra constraints) in the first RMP (RDM) that, combined with the ordinary variables, lead the RMP to implicitly consider variables (associated with extreme points of the subproblem) that are not explicitly present in the RMP. In the same reference that approach is applied in a cutting stock problem, where the extra variables are associated with replacing a larger item with two smaller ones. As an example, taking a set of patterns/columns in which a specific large item belongs, those extra columns allow the representation of all the patterns/columns derived from the above mentioned replacement of items, avoiding their generation by the subproblem.

Those extra variables are kept in all iterations and, when there are no more attractive columns, a solution where they all have zero value is recovered by a problem specific procedure. Another option, in order to obtain an optimal solution expressed exclusively on the ordinary variables, is to slightly penalise the extra variables (Amor et al., 2003).

## 2.5 Dantzig-Wolfe Decomposition in Integer Programming

### 2.5.1 Branch-and-price overview

A fundamental difference when dealing with integer problems (as opposed to linear ones) is that no general necessary or sufficient optimality conditions are known. Two families of methods have been employed to exactly solve integer programming problems: branch-and-

bound methods and cutting plane methods. Both methods use relaxations in order to derive lower bounds (in a minimisation problem) to the value of an optimal solution. A fundamental issue is the quality of the lower bounds of the relaxations.

Under certain circumstances, the lower bounds provided by the relaxation of the DWD (re)formulation are better than the provided by the linear relaxation. This is a main motivation for the use of column generation based algorithms / DWD in integer programming problems (but not neglecting the ones mentioned in Section 2.1). This subject is detailed in subsection 2.5.2.

Branch-and-price algorithms combine column generation and branch-and-bound in order to obtain (optimal) solutions to integer programming problems. Columns that were not generated in the root node of the branch-and-bound tree may be required in an optimal integer solution, and thus in every node of the branch-and-bound (branch-and-price) tree it may be necessary to generate columns, if an optimal solution is desired. This requires compatibility between the branching rules and the subproblem, which is the subject of subsection 2.5.3.

The perspective taken here is that, in each node of the branch-and-price tree, the original formulation is implicitly considered, meaning that it is always possible to represent in the (restricted) master problem a constraint expressed in terms of the original variables. Being so, cuts expressed in the original variables can be added to the RMP. As long as they are kept in the RMP, the feasible region of the subproblem is not changed, allowing the easy incorporation of cuts in the branch-and-price algorithm, thus obtaining a branch-and-price-and-cut algorithm. This subject is detailed in subsection 2.5.4.

In subsection 2.5.5, the simultaneous definition of different subproblems for the same original formulation and its implications are considered.

Subsection 2.5.6 is devoted to other relevant issues of branch-and-price and to a brief discussion of a different perspective that leads to other type of branch-and-price algorithms.

### 2.5.2 Lower bounds given by the Dantzig-Wolfe decomposition

For clarity of exposition and simplicity of notation, in this Section, except where clearly stated otherwise, we consider the general pure integer programming model:

$$\begin{aligned} &Min\ c\,x && (PIP)\\ &subject\ to:\\ &D\,x \geq d\\ &A\,x = b && (2.14)\\ &x \geq 0 && (2.15)\\ &x\ integer. && (2.16) \end{aligned}$$

The dimensions of the matrices are the same as the ones in *(LnP)* (introduced in Section

2.2.2, page 15).

By defining the subproblem through constraints *(2.14)*, *(2.15)*, and *(2.16)*, its feasible region is defined by $S_{SPI} = \{ x : Ax = b, x \geq 0, x \text{ integer} \}$, the master of a DWD will be:

$$Z_{LDWI} = \quad Min \sum_{p \in P} (c\, y^p)\, \lambda_p + \sum_{r \in R} (c\, u^r)\mu_r \qquad (LDWI)$$

*subject to:*

$$\sum_{p \in P} (D\, y^p)\lambda_p + \sum_{r \in R} (D\, u^r)\mu_r \geq d$$

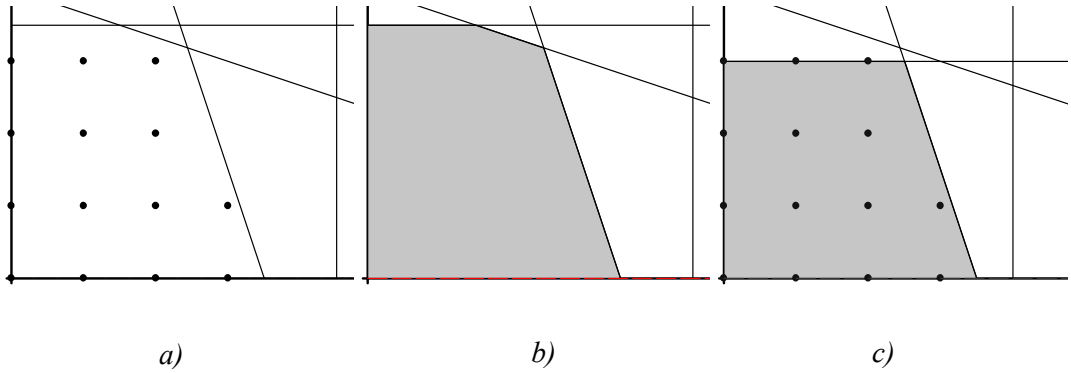
$$\sum_{p \in P} \lambda_p = 1$$

$$\lambda_p \geq 0, \ \forall p \in P^I$$

$$\mu_r \geq 0, \ \forall r \in R^I,$$

where the sets $P^I$ and $R^I$ are associated with the extreme points and extreme rays of $S_{SPI}$, where, by definition, all the original $x$ points have integer values.

As illustrated in Figure 2.14, by considering the integrality constraints in the subproblem, its feasible region is reduced, which leads to the reduction of the feasible region of the master problem (since it consists in all possible convex combinations of the extreme points of the subproblem). The previous statement is valid if the subproblem has non-integer extreme points; otherwise the subproblem has the integrality property, that is $S_{SP} = S_{SPI}$. If the subproblem does not have the integrality property, the application of the DWD leads to a tighter relaxation of the integer problem and, for some objective functions, the relation $Z_{LDWI} > Z_{LDW} = Z_{LnP}$ may hold. In general, $Z_{LDWI} \geq Z_{LDW} = Z_{LnP.}$



*a)* *b)* *c)*

Figure 2.14 Illustration of the (possibly) different lower bounds given by the linear relaxation and by the DWD (re)formulation. *a)* Original integer problem. *b)* Linear relaxation. *c)* DWD (re)formulation defining the subproblem through the upper bound and integrality constraints (represented in the original solution space).

As noted when discussing the use of DWD in linear programming, different subproblems for the same model can be defined. As an example, in Figure 2.15 the same problem of Figure

2.14a) is considered, but with the definition of the subproblem with all the constraints other than the upper bound constraints.

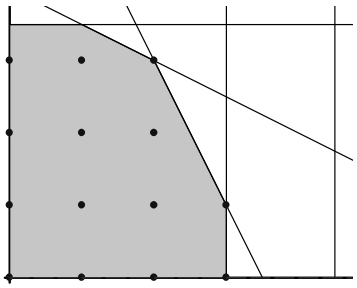


Figure 2.15 Illustration of a different (from Figure 2.14) choice of the constraints defining the subproblem.

This is a main issue when defining the decomposition approach to solve a(n) (integer) problem: which constraints should define the subproblem?

On one hand, since the subproblem is (re)optimised a large number of times, it is important to have subproblems with some kind of structure for which efficient algorithms are known. On the other hand, the definition of the subproblem should lead to good quality lower bounds, thus being "as far as possible" from the integrality property, what, in general, increases its difficulty. In addition, the difficulty of solving the resulting RMPs can have an important impact in the practical efficiency of a given decomposition.

Getting the right balance between the aforementioned issues clearly requires a problem dependent approach. The binary multicommodity flow problem studied in Chapter 4 is an example of how two different decompositions of the same original model may have totally different characteristics.

We end this subsection by noting that all the methods developed in the last decades, particularly those briefly described in subsection 2.3.4, to solve the Lagrangean dual give the same bound as the DWD principle when used in integer programming. However, with the exception of subgradient methods (with its inherent disadvantages, already pointed out), their use in integer programming is confined to a few experiments (see, for example, (Cappanera and Frangioni, 2000; Elhedhli and Goffin, 2001)).

### 2.5.3 Branching rules

Solving the problem *(LDWI)* by column generation gives a lower bound that is not worse than the one given by the linear relaxation of the original formulation *(PIP)*.

We recall the relation between the original variables, $x$, and the weight variables of the master model, $\lambda$ and $\mu$,

$$x = \sum_{p \in P^I} y^p \lambda_p + \sum_{r \in R^I} u^r \mu_r$$

$$\sum_{p \in P^I} \lambda_p = 1$$

$$\lambda_p \geq 0, \ \forall p \in P^I$$

$$\mu_r \geq 0, \ \forall r \in R^I.$$

From this relation, it is clear that if all the weight variables have integer values in the optimal solution of the root of the branch-and-price tree, *(LDWI)*, an optimal solution to the integer problem has been found, since they are associated with integer extreme points and rays.

A formulation to the integer problem is obtained by adding integrality constraints on the variables of *(LDWI)*. In order to solve that problem exactly, branching on the weight variables may be required. In a simple branching scheme, two new problems are constructed by adding a constraint on a fractional variable (indexed by $\bar{p}$) to each one,

$$\lambda_{\bar{p}} \leq \lfloor \bar{\lambda}_{\bar{p}} \rfloor \text{ and } \lambda_{\bar{p}} \geq \lfloor \bar{\lambda}_{\bar{p}} \rfloor + 1,$$

where $\bar{\lambda}_{\bar{p}}$ denotes the current value of the fractional variable.

In general, this branching scheme has a major disadvantage: its implementation without changing the structure of the subproblem may not be easy. For example, if the (binary) master amounts to combining paths given by a subproblem that is solved using a shortest path problem, in one descendant node a given path is excluded and, in the other, the same path is forced to be included in the solution. Excluding a given path from being the optimal solution of a shortest path problem is not a trivial problem, and for sure requires substantial modifications on the algorithm to solve the subproblem, since the extreme point that should be kept out of the (restricted) master problem may be generated by the subproblem. This issue is usually called "regeneration".

A possibility to overcome this problem is to solve the shortest path problem and, if the path that should be excluded is the optimal one, then to find the second best path. However, in a node of the search tree where *k* paths must be excluded, this approach may lead to solve the *k*-shortest paths problem, making the subproblem much more difficult to solve than the one of the root node.

Another disadvantage of this branching scheme is that it may lead to unbalanced search trees. Taking again a binary problem as an example, branching is performed in a single variable and the problem (in general) has a very large number of variables, thus it is more likely that the optimal solution is in the branch $\lambda_{\bar{p}} = 0$ then in the other one, $\lambda_{\bar{p}} = 1$. Examples of these type of branching schemes are the ones of (Ribeiro et al., 1989) and (Park et al., 1996).

Branching schemes, based on the original variables, usually overcome these difficulties.

After obtaining the optimal solution of a node, it is always possible to express it in the original variables through

$$\bar{x} = \sum_{p \in \bar{P}^I} \bar{\lambda}_p y^p + \sum_{r \in \bar{R}^I} \bar{\mu}_r u^r ,$$

where $\bar{P}^I$ and $\bar{R}^I$ denote the set of indices of the extreme points and rays, respectively, associated with the columns of the current RMP.

Branching on a fractional original variable, the element $\bar{x}_j$ of the vector $\bar{x}$, can be done creating two descendant problems constructed by adding, respectively, the following constraints

$$\sum_{p \in \bar{P}^I} \bar{\lambda}_p y_j^p + \sum_{r \in \bar{R}^I} \bar{\mu}_r u_j^r \leq \lfloor \bar{x}_j \rfloor \text{ and } \sum_{p \in \bar{P}^I} \bar{\lambda}_p y_j^p + \sum_{r \in \bar{R}^I} \bar{\mu}_r u_j^r \geq \lfloor \bar{x}_j \rfloor + 1 ,$$

where the index *j* refers to the position *j* of the vectors defining the extreme points and rays.

If the branching constraints are kept in the master problem, the modifications implied in the subproblem are confined to its objective function, as it is clear by noting that in a node of a branch-and-bound tree, we have the following problem in the original variables:

$$\begin{aligned}
&\textit{Min } c\,x \\
&\textit{subject to:} \\
&D\,x \geq d \\
&G\,x \geq g && (2.17)\\
&A\,x = b && (2.18)\\
&x \geq 0 && (2.19)\\
&x \text{ integer,} && (2.20)
\end{aligned}$$

where the matrix *G* and the vector *g* represent the coefficients of the branching constraints *(2.17)*.

Thus, defining the subproblem as previously (by constraints *(2.18)*, *(2.19)*, and *(2.20)*):

$$\begin{aligned}
&\textit{Min } (c - \bar{w}\,D - \bar{w}\,G)\,x \\
&\textit{subject to:} \\
&x \in S_{SPI} .
\end{aligned}$$

This branching scheme is general. We considered a pure integer problem for easiness of notation and exposition, but the extension to mixed integer problems (that may have simultaneously linear, binary and integer variables) is done easily.

For binary problems, it is possible to use a different branching scheme, also based on the original variables, that consists in forcing the branching decisions in the subproblems, performing minor amendments in the RMP.

In this case the subproblem in a node of a branch-and-price tree is

$$Min\ (c - \overline{w}\, D)\, x$$
$$x \in S_{SPI}$$
$$x_j = 0,\ \forall j \in L$$
$$x_j = 1,\ \forall j \in U,$$

where $L$ and $U$ are the sets of indices of variables for which a branching constraint (forcing the variable to *0* or *1*, respectively) exists.

In the RMP, forcing an original variable to take value *0* implies removing all columns associated with the extreme points and rays in which that original variable has value *1*. In the same manner, forcing an original variable to take value *1* implies removing all columns associated with the extreme points and rays in which that original variable has value *0*. In both cases, the subproblem assures that none of those columns, nor the columns with the same characteristics, will be generated.

As an example, consider a master problem that amounts to combining binary knapsack solutions. Each column is associated with a solution to the binary knapsack (sub)problem. Forcing an item out of the knapsack is done by removing all columns where it is included in the knapsack and by solving the subproblem without the item. Forcing an item in the knapsack is done by removing all columns where it is not included in the knapsack and by solving the subproblem considering that the item is in the knapsack. In both cases the subproblem structure does not change due to the branching. With this branching scheme, that is not always the case. Taking again the example of a (binary) master problem that amounts to combining paths, excluding an arc from the (shortest path) subproblem does not involve major changes in its structure. However, forcing an arc to be included in a path changes the structure of the subproblem, which becomes a *set* of shortest path problems. This issue can be overcome by using branching rules specific to the problem in question, thus without the generality of the previous approach, where branching constraints are kept in the master.

We end this subsection by noting that the master model of an important set of problems where branch-and-price algorithms have been successful applied corresponds to set partition problems (Desrochers and Soumis, 1989; Desrochers et al., 1992; Vance et al., 1997; Savelsbergh and Sol, 1998) for which specific branching rules were devised, such as the Ryan and Foster (described, for example, in (Wolsey, 1998)).

### 2.5.4 Branch-and-price-and-cut

The fundamental idea of cutting plane methods is to tighten the linear relaxation of an integer problem through the introduction of valid inequalities (cuts). Those valid inequalities may be (i) constraints that are intentionally left out of the original formulation because their number is exponential (for example, the subtour elimination constraints for the travelling

salesman problem), (ii) general valid inequalities (for example, Gomory cuts) and (iii) specific valid inequalities (for example, the ones pointed out in (Wolsey, 2002) for several lot sizing problems).

Since, in general, a class(es) of cuts that define the integer convex hull is not known (or solving the separation problem repeatedly for identifying cuts that exclude the current fractional solution may be too costly), branch-and-cut algorithms are based on performing branching as soon as no violated cuts can be (at all, or efficiently) identified, thus combining cutting planes with branch-and-bound. Branch-and-cut algorithms have their roots in (Crowder et al., 1983) and (Padberg and Rinaldi, 1991). A recent survey is (Marchand et al., 2002).

Cutting plane methods can also be combined with branch-and-price, giving rise to branch-and-price-and-cut algorithms.

As detailed in the previous subsection, additional constraints (there branching constraints, here cuts) expressed in the original variables can be easily expressed in weight variables and added to the RMP, only changing the coefficients in the objective function of the subproblem.

The problem of a node of a branch-and-price-and-cut tree expressed in the original variables is

$$\begin{aligned}
&Min\ c\,x \\
&subject\ to: \\
&D\,x \geq d && (2.21)\\
&G\,x \geq g \\
&H\,x \geq h && (2.22)\\
&A\,x = b && (2.23)\\
&x \geq 0 && (2.24)\\
&x\ integer, && (2.25)
\end{aligned}$$

where the matrix $H$ and the vector $h$ represent the coefficients of the cuts *(2.22)*.

Thus, defining the subproblem as previously (by constraints *(2.23)*, *(2.24)*, and *(2.25)*):

$$\begin{aligned}
&Min\ (c - \overline{w}\,D - \overline{w}\,G - \overline{w}\,H)\,x \\
&subject\ to: \\
&x \in S_{SPI}.
\end{aligned}$$

Solving a node of the branch-and-price-and-cut tree consists in the following sequence of steps.

1. Apply column generation to obtain a solution in the weight variables.
2. Express the obtained solution in the original variables.
3. Solve a separation problem for the solution expressed in the original variables.
4. If a cut is found, add it to the RMP and go to 1. Else, stop.

We end this subsection by pointing out that the convex hull of the integer polyhedron defined by the subproblem constraints is already described, and thus only classes of cuts for defining the convex hull of the polyhedron defined by the integer intersection of the linking constraints *(2.21)* with the subproblem constraints, *(2.23)*, *(2.24)*, and *(2.25)*, are of interest for a branch-and-price-and-cut method.

### 2.5.5 Multiple Dantzig-Wolfe decomposition

To our best knowledge, the idea of using several Dantzig-Wolfe decompositions simultaneously has appeared recently (Aragão and Uchoa, 2003; Park et al., 2003).

In the first reference, the authors develop a formulation for the binary multicommodity flow problem that leads to two types of variables in the master problem according to two different subproblems. That formulation is directly derived for the specific problem (not turning explicit the original model and the decompositions being used) and the paper is not concerned on how to extend that kind of approach to a general problem. For the specific problem treated in that reference, the possible advantages of this approach are clear: one subproblem has the integrality property, being easier to solve, and captures the network structure of the original problem, while the other does not have the integrality property, but leads to better quality lower bounds.

In the second reference, the authors develop what they call a robust branch-and-price methodology. The main difference between this approach and the one described before in this Chapter is that the relation between the original variables and the weight variables is explicitly defined in the master problem by a set of equality constraints. Being so, the (extended) master problem is

$$\begin{aligned}
&\textit{Min } c\,x \\
&\textit{subject to:} \\
&x = \sum_{p \in P} \lambda_p x^p + \sum_{r \in R} \mu_r u^r && (2.26) \\
&\sum_{p \in P} \lambda_p = 1 && (2.27) \\
&A\,x \geq b \\
&x \geq 0 \\
&\lambda_p \geq 0, \ \forall p \in P \\
&\mu_r \geq 0, \ \forall r \in R.
\end{aligned}$$

As discussed in the two previous subsections, the absence of the original variables in the master problem does not affect the ability to perform branch and/or cut in their space, as long as the constraints are represented in the space of the weight variables. The approach taken here has

the clear advantage of dealing with smaller (restricted master) problems and is also general (robust in the terminology used by the authors), given the relation between original and weight variables.

In the same reference, based on the extended formulation given above, the authors outline multiple column generation. It amounts to defining more than one subproblem, having more than one type of columns in the master formulation. Since the original variables, at the expense of having a larger problem, are present in that formulation, it is easy to state that the convex combination of the extreme points (in the bounded case) of each subproblem must be equal to the original variables, by reproducing constraints *(2.26)* and *(2.27)* for each subproblem.

Our contribution is to develop multiple column generation without explicitly having the original variables in the master.

We consider the problem

$$
\begin{aligned}
Z_{MPIP} = \quad & Min\ c\,x && (MPIP)\\
& subject\ to: &&\\
& D\,x = d && (2.28)\\
& A\,x = b && (2.29)\\
& x \geq 0 &&\\
& x\ integer. &&
\end{aligned}
$$

For simplicity of notation, we consider only two subproblems with feasible regions defined by:

$S_{MSP1} = \{ x : Dx = d, x \geq 0, x\ integer\}$ and

$S_{MSP2} = \{ x : Ax = b, x \geq 0, x\ integer\}$.

Clearly, problem *(MPIP)* is equivalent to

$$
\begin{aligned}
& Min\ c\,x^1\\
& subject\ to:\\
& x^1 \in S_{MSP1}\\
& x^2 \in S_{MSP2}\\
& x^1 = x^2\\
& x^1\ integer.
\end{aligned}
$$

A solution in the space of the original model *(MPIP)* can be expressed as a convex combination of the extreme points plus a nonnegative combination of the extreme rays of $S_{MSP1}$, as well as a convex combination of the extreme points plus a nonnegative combination of the extreme rays of $S_{MSP2}$:

| *SP1* | *SP2* |
|---|---|
| $x^1 = \sum_{p1 \in P^1} \lambda_{p1} y^{p1} + \sum_{r1 \in R^1} \mu_{r1} u^{r1}$ | $x^2 = \sum_{p2 \in P^2} \lambda_{p2} y^{p2} + \sum_{r2 \in R^2} \mu_{r2} u^{r2}$ |
| $\sum_{p1 \in P^1} \lambda_{p1} = 1$ | $\sum_{p2 \in P^2} \lambda_{p2} = 1$ |
| $\lambda_{p1} \geq 0, \ \forall p \in P^1$ | $\lambda_{p2} \geq 0, \ \forall p \in P^2$ |
| $\mu_{r1} \geq 0, \ \forall r \in R^1,$ | $\mu_{r2} \geq 0, \ \forall r \in R^2.$ |

The master problem (neglecting integrality constraints imposed by branching on the original variables) will be:

$$Min \sum_{p1 \in P^1} (c\, y^{p1})\, \lambda_{p1} + \sum_{r1 \in R^1} (c\, u^{r1}) \mu_{r1}$$

*subject to:*

$$\sum_{p1 \in P^1} \lambda_{p1} = 1 \qquad (2.30)$$

$$\sum_{p2 \in P^2} \lambda_{p2} = 1 \qquad (2.31)$$

$$\sum_{p1 \in P^1} \lambda_{p1} y^{p1} + \sum_{r1 \in R^1} \mu_{r1} u^{r1} = \sum_{p2 \in P^2} \lambda_{p2} y^{p2} + \sum_{r2 \in R^2} \mu_{r2} u^{r2} \qquad (2.32)$$

$\lambda_{p1} \geq 0, \ \forall p1 \in P^1$

$\lambda_{p2} \geq 0, \ \forall p2 \in P^2$

$\mu_{r1} \geq 0, \ \forall r1 \in R^1$

$\mu_{r2} \geq 0, \ \forall r2 \in R^2.$

Constraints *(2.30)* and *(2.31)* assure feasibility of the original constraints *(2.28)* and *(2.29)*, respectively. Constraints *(2.32)* assure that the same original point is being considered in both subproblems.

The subproblems are:

| *SP1* | *SP2* |
|---|---|
| $Min\ (c - w)\, x - \pi^1$ | $Min\ w\, x - \pi^2$ |
| *subject to:* | *subject to:* |
| $x \in S_{MSP1}$, | $x \in S_{MSP2}$, |

where $w$ are the duals of constraints *(2.32)*, and $\pi^1$ and $\pi^2$ are the duals of constraints *(2.30)* and *(2.31)*, respectively.

Branching and/or cutting can still be performed as previously described by taking the original variables. Taking branching as example, branching on a fractional original variable, the

element $\bar{x}_j$ of the vector $\bar{x}$, can be done creating two descendant problems constructed by adding, respectively, the following constraints

$$\sum_{p1 \in \bar{P}^1} \bar{\lambda}_{p1} y_{j1}^{p1} + \sum_{r1 \in \bar{R}^1} \bar{\mu}_{r1} u_{j1}^{r1} \leq \lfloor \bar{x}_j \rfloor \text{ and}$$

$$\sum_{p1 \in \bar{P}^1} \bar{\lambda}_{p1} y_{j1}^{p1} + \sum_{r1 \in \bar{R}^1} \bar{\mu}_{r1} u_{j1}^{r1} \geq \lfloor \bar{x}_j \rfloor + 1,$$

where the indices *j1*, *p1*, and *r1*, denote that the branching is being performed on the representation given by the subproblem 1 of the original variables.

This approach is the dual of Lagrangean decomposition, introduced in (Guignard and Kim, 1987). In that reference the emphasis is on obtaining good quality lower bounds, which may be better than the ones given by standard Lagrangean relaxation if more than one subproblem does not have the integrality property.

We end this subsection giving a small numerical example of multiple DWD / column generation.

**Example 2.3**

For illustration of multiple DWD and the column generation procedure to solve it, we give a very simple example based on the original problem:

$$Min\ -2x_1 - x_2$$

*subject to:*

$$x_1 + x_2 \leq 3$$

$$x_1 \leq 2$$

$$x_2 \leq 2$$

$$x_1, x_2 \geq 0.$$

In Figure 2.16, the feasible region is depicted.

We define SP1 through the first constraint and SP2 through the other two. We consider a first RMP where the only one extreme point – the origin – of each subproblem is present. All the dual variables have zero value in the optimal solution, thus we solve the subproblems:

| *SP1* | *SP2* |
|---|---|
| *Min* $-2x_1 - x_2$ | *Min 0* |
| *subject to:* | *subject to:* |
| $x_1 + x_2 \leq 3$ | $x_1 \leq 2$ |
| $x_1, x_2 \geq 0,$ | $x_2 \leq 2$ |
| | $x_1, x_2 \geq 0.$ |

Optimal solutions are $x_1=3$, $x_2=0$, and $x_1=2$, $x_2=2$, for subproblems 1 and 2, respectively. Thus, the RMP becomes

$$
\begin{aligned}
&Min\ -6\lambda_{21} \\
&subject\ to: \\
&\lambda_{11} + \lambda_{21} = 1 \\
&\lambda_{12} + \lambda_{22} = 1 \\
&3\lambda_{21} - 2\lambda_{22} = 0 \\
&-2\lambda_{22} = 0 \\
&\lambda_{11}, \lambda_{12}, \lambda_{21}, \lambda_{22} \geq 0,
\end{aligned}
$$

with optimal value *0* and duals $\pi^1=0$, $\pi^2=0$, $w_1=-2$, and $w_2=2$.

The only primal solution that is feasible to the RMP remains $x_1=x_2=0$, as can be seen in Figure 2.17, where the solutions of each subproblem that are being considered in the RMP are depicted. The subproblems are now

| *SP1* | *SP2* |
|---|---|
| *Min* $-3x_2$ | *Min* $-2x_1 + 2x_2$ |
| *subject to:* | *subject to:* |
| $x_1 + x_2 \leq 3$ | $x_1 \leq 2$ |
| $x_1, x_2 \geq 0,$ | $x_2 \leq 2$ |
| | $x_1, x_2 \geq 0.$ |

Their optimal solutions are $x_1=0$, $x_2=3$, and $x_1=2$, $x_2=0$, respectively. The RMP becomes

$$
\begin{aligned}
&Min\ -6\lambda_{21} - 3\lambda_{31} \\
&subject\ to: \\
&\lambda_{11} + \lambda_{21} + \lambda_{31} = 1 \\
&\lambda_{12} + \lambda_{22} + \lambda_{32} = 1 \\
&3\lambda_{21} - 2\lambda_{22} - 2\lambda_{32} = 0 \\
&-2\lambda_{22} + 3\lambda_{31} = 0
\end{aligned}
$$

$$\lambda_{11},\ \lambda_{12},\ \lambda_{21},\ \lambda_{22},\ \lambda_{31},\ \lambda_{32} \geq 0,$$

with optimal value *0* and duals $\pi^1=-3$, $\pi^2=-2$, $w_1=-1$, and $w_2=0$. The subproblems are now

| *SP1* | *SP2* |
|---|---|
| $Min\ -x_1 - x_2 + 3$ | $Min\ -x_1 + 2$ |
| *subject to:* | *subject to:* |
| $x_1 + x_2 \leq 3$ | $x_1 \leq 2$ |
| $x_1, x_2 \geq 0,$ | $x_2 \leq 2$ |
| | $x_1, x_2 \geq 0.$ |

Both optimal objective values are zero, thus the current optimal solution of the RMP is optimal to the overall problem: $\lambda_{21}=2/3$, $\lambda_{22}=0.5$, $\lambda_{31}=1/3$, $\lambda_{32}=0.5$ with value $-5$, or, in the original variables $x_1 = (2/3).3 + (1/3).0 = (0.5).2 + (0.5).2 = 2$ and $x_2 = (2/3).0 + (1/3).3 = (0.5).2 + (0.5).1 = 1$.

The feasible region of the last RMP (in the original space) is depicted in Figure 2.18.

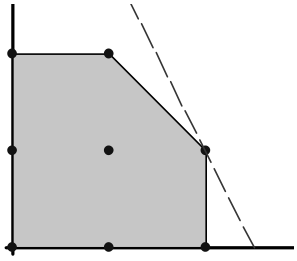


Figure 2.16 Feasible region of the problem of Example 2.3.

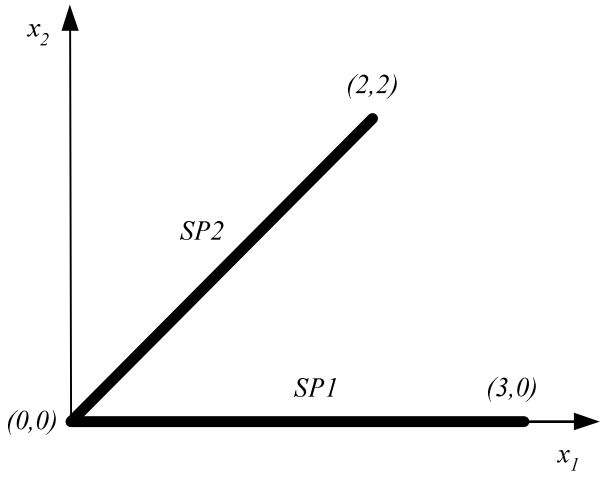



Figure 2.17 Feasible region of the second RMP of Example 2.3.

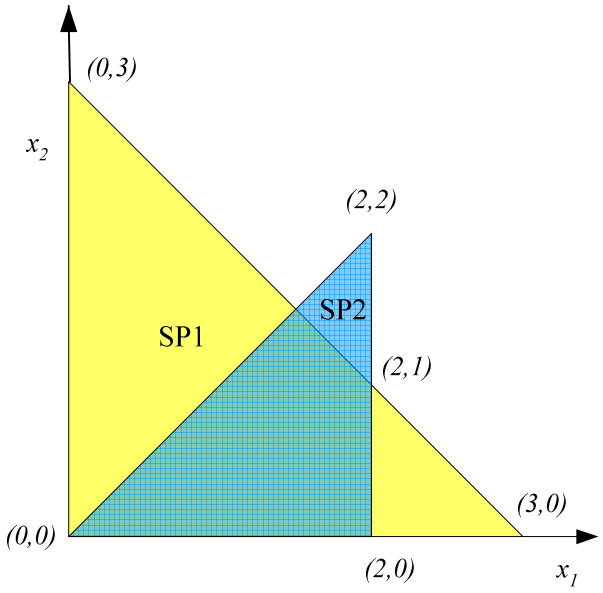



Figure 2.18 Feasible region of the last RMP of Example 2.3.

♦

### 2.5.6 Relation with standard branch-and-bound and a related approach

Taking a conceptual perspective, by defining branching rules and cuts on the original variables, by expressing them in the weight variables afterwards, and by inserting them in the RMP, branch-and-price(-and-cut) is nothing more than branch-and-bound (with cuts) where each node is solved by column generation.

This perspective can be further explored to include preprocessing and variable fixing. For example, after optimising a RMP, if the reduced cost of a nonbasic binary variable plus the optimal value of the RMP is larger than the incumbent, the variable can be fixed at zero. Preprocessing and variable fixing in branch-and-price is discussed in (Vanderbeck, 2005).

When solving a node of the branch-and-price(-and-cut) tree, the lower bound given in subsection 2.2.4 (page 23) is available in each iteration of column generation. If that lower bound is larger than the incumbent value, then the node can be pruned, since the best integer solution found in it and in its descendants (if any) will have a larger value than the incumbent.

Primal heuristics can also be incorporated in branch-and-price. Since the original formulation is known, it can be used to provide upper bounds, or even feasible primal integer solutions as columns in the RMP.

In this Section, we took the convexification approach on branch-and-price. That is, the

master problem was obtained by representing solutions as combinations of the extreme points and rays of the convexified subproblem. This approach allows solving each node of the branch-and-price tree in the same manner as the root node: there is no change on the subproblem structure (except that the calculation of the coefficients of its variables must take into account the duals of the branching and cut constraints) by keeping the branching and cut constraints in the master.

We would like to point out a different approach that relies in the discretisation of the subproblem (Vanderbeck and Wolsey, 1996; Vanderbeck, 2000). In that case, the master solutions are represented as a combination of a finite set of integer points (not necessarily extreme) and a finite set of integer rays of the subproblem. For binary problems both approaches are equivalent (since the subproblem does not have interior integer solutions) but for general integer problems this second approach may give better lower bounds and provide more elaborate branching rules, namely for dealing with symmetry (that is, ineffective branching due to the fact that the current solution is excluded but a similar one with the same value and meaning is not). Those issues are discussed in the references above given.

## 2.6 Conclusions

In this Chapter, Dantzig-Wolfe decomposition and column generation based algorithms for linear and integer programming problems were surveyed.

Motivations for using decomposition approaches, references to applications in several areas and the fundamental theory and algorithmic aspects were given.

The close relation between Dantzig-Wolfe decomposition and Lagrangean relaxation, and the relation between column generation (and variants) and methods for solving the Lagrangean dual were made explicit.

The application of column generation based algorithms in integer programming was also reviewed, with a focus on general branching schemes and the inclusion of cuts (branch-and-price-and-cut).

Multiple Dantzig-Wolfe decomposition and multiple column generation was introduced in a general way, compatible with a general branching scheme and with the use of cuts.

Several recent references were given (some for detailed treatment of topics not covered here), proving, after more than forty years of the publication of the Dantzig-Wolfe decomposition principle and of development of the first column generation based algorithms, their actual relevance (and not fully explored potential) for dealing with optimisation problems.

# 3 Integer Multicommodity Flow Problem

In this Chapter, we develop a branch-and-price algorithm for the integer minimum cost multicommodity flow problem.

This problem is defined over a directed network in which several commodities share the capacity of the arcs, in order to be shipped from the nodes in which they are supplied to the nodes in which they are demanded. Commodities may have several origin and destination nodes.

Although several approaches have been described for its linear version, the same does not happen for the integer problem considered here. The extension of those approaches is not trivial, particularly when using path based formulations, which may be the only feasible approach, due to computational memory limits, given the generally much larger size of arc based formulations.

We develop a branching rule that preserves the structure of the subproblem in the nodes of the branch-and-price tree by considering cycle variables, not needed when solving the linear relaxation. The same type of approach can be used in related network flow problems.

We present computational results for the proposed algorithm and for a general purpose solver.

## 3.1 Introduction

The subject of this Chapter is the minimum cost integer multicommodity flow problem (MFP), for which a branch-and-price algorithm is presented. This problem is defined over a directed network in which several commodities share the capacity of the arcs, in order to be shipped from their origin to their destination nodes. Associated with each arc of the network and with each commodity there is a unit flow cost. The minimum cost integer MFP amounts to finding the minimum cost routing of all the commodities, taking into account that each unit of each commodity cannot be split.

The linear version of this problem (where units can be split) has deserved the interest of the Operational Research and close scientific communities for more than forty years, since the pioneering work of Ford and Fulkerson on network flows (Ford and Fulkerson, 1962). This interest has been continuous: at least one survey has been published in each of the last five decades (Hu, 1963; Assad, 1978; Kennington, 1978; Kennington and Helgason, 1980; Ahuja et al., 1993; Chardaire and Lisser, 2002a).

This interest can be justified by the practical and theoretical importance of multicommodity flow models. From the practical side, their relevance is based on their many applications, such as communications systems (for an excellent annotated bibliography, see (Yuan, 2001)), production planning (for example, (Evans, 1977; Zahorik et al., 1984)) and distribution/transportation (for example, (Desrosiers et al., 1995)). Other applications of the linear minimum cost MFP can be found in the surveys already mentioned.

From the theoretical side, multicommodity flow models are representative of large linear programs with block-angular structure with linking constraints, being a source of inspiration and testing for new decomposition methods, from the Dantzig-Wolfe decomposition (DWD) principle (Dantzig and Wolfe, 1960) (which was inspired by the work of Ford and Fulkerson on the maximal multicommodity flow problem (Ford and Fulkerson, 1958), as mentioned in (Dantzig, 1963)) to specialised interior-point methods (Schultz and Meyer, 1991).

As already mentioned, this Chapter is devoted to the minimum cost integer MFP. This problem has received considerably less attention than its linear version, although in several applications it may be an important issue to consider that the units of the commodities being routed are unsplittable. We note that this problem is different from the binary MFP, in which a commodity must be routed along a single path (and thus, each commodity has only one origin and one destination). In Chapter 4, the same approach presented in this Chapter will be used to tackle that particular binary problem.

Our approach is based on using column generation on a formulation based on flows in paths combined with branch-and-bound (resulting in a branch-and-price method). Column generation allows solving the linear relaxation of the integer MFP in a very efficient way, so its combination with branch-and-bound is a promising approach to solve the integer MFP.

Although column generation and branch-and-bound are known for about four decades, the potential of their combination for obtaining optimal integer solutions became clear only in recent years. Branch-and-price methods were reviewed in Chapter 2; other surveys can be found in (Barnhart et al., 1998; Wilhelm, 2001; Lübbecke and Desrosiers, 2002).

A major motivation for the development of a branch-and-price algorithm is the quality of the lower bounds given by the underlying Dantzig-Wolfe reformulation. However, that motivation is irrelevant in the present work, since the subproblem (a set of independent shortest path problems) has the integrality property. Our main motivation is the potential efficiency of a branch-and-price algorithm for the integer MFP, justified by the use of a decomposition that captures the (network) structure of the problem.

The literature on branch-and-price algorithms for general integer variables is still scarce, as opposed to the one with binary variables. Exceptions are (Vanderbeck and Wolsey, 1996; Carvalho, 1998; Vance, 1998; Carvalho, 1999; Vanderbeck, 1999; Vanderbeck, 2000), with experiments in the cutting stock problem.

Besides providing an algorithm for integer MFP (that can be extended to other multicommodity flow problems), we aim at exploring the branch-and-price method for problems with a network structure. In particular, by developing a branching scheme that may be considered in other types of problems. The innovative aspect of this approach is that it overcomes the possible generation of negative cycles in the shortest path (sub)problems. Although this issue has a clear interpretation in the DWD context, to our best knowledge, it has never been treated before.

We now give examples where the approach presented here can be used for solving related problems, also noting that in some of the applications, it may be worth considering that the units of the commodities being routed are unsplittable.

Here we consider that the arcs of the network are oriented. In the non-oriented version of the problem, the capacity of each arc limits the flow in both directions, and thus a different formulation must be considered. The method presented here can easily be extended to that closely related problem.

In the maximum multicommodity flow, the arcs of the network and their capacities are known and a maximum flow of all commodities (from their origins to their destinations) is desired (Ford and Fulkerson, 1958; Kennington, 1978; Kapoor and Vaidya, 1996). The

approach presented here can be easily used in obtaining optimal solutions to that problem.

As an example of a problem where the linear multicommodity flow can be used as a subproblem, we present the simple network design problem, where a selection of the arcs, each one associated with a fixed cost, must be performed with the objective of minimising the total cost (fixed cost plus routing cost on the selected arcs) (Magnanti and Wong, 1984; Minoux, 1989; Gendron et al., 1999). For a fixed configuration of the network, the routing problem is a MFP.

Taking into account the congestion of the arcs, which typically is modelled by concave functions, a nonlinear minimum cost multicommodity flow problem is obtained (Ouorou et al., 2000). The algorithm presented here can be seen as a combination of branch-and-bound with a cutting plane method that can be applied to this concave problem by changing the RMP solver.

We now outline the contents of this Chapter. In Section 3.2, we present a formal definition of the integer MFP and introduce three formulations for that problem. A review of solution methods to the linear relaxation of the integer MFP is also given in that Section. In Section 3.3, the proposed branch-and-price algorithm is presented. Two different branching rules and their consequences in the structure of the subproblem are discussed. The need for the cycle variables in the nodes of the branch-and-price tree is clarified. In Section 3.4, we describe some implementation issues and compare the computational performance of different versions of the algorithm for some instances. We also compare, for several publicly available sets of instances, the performance of our branch-and-price program with a program that uses Cplex callable library 6.6 (ILOG, 1999) to solve the arc formulation. In Section 3.5, we present the main conclusions of this work.

## 3.2 Formulations and Review of Solution Methods

### 3.2.1 Problem definition and arc formulation

We consider a network defined by a set of $n$ nodes, denoted by $N$, and a set of $m$ directed arcs, denoted by $A$. We also consider a set of $h$ commodities, denoted by $K$. Associated with each arc $ij \in A$, there is an origin node $i$ and a destination node $j$. Associated with each node $i$ and with each commodity $k$, there is a parameter $b_i^k$. If $b_i^k > 0$, node $i$ is an origin to commodity $k$, with supply of $b_i^k$ units. If $b_i^k < 0$, node $i$ is a destination to commodity $k$, with demand of $-b_i^k$ units. If $b_i^k = 0$, the node is a transhipment node to commodity $k$.

Associated with each arc $ij$ and with each commodity $k$ there is a parameter $c_{ij}^k$ that corresponds to the unit flow cost of that commodity in that arc. The usual assumption, $c_{ij}^k \geq 0$, $\forall ij \in A$, $\forall k \in K$, is made.

Each arc $ij$ has a capacity $u_{ij}$, which is the limit to the total flow in the arc. Supplies, demands and arc capacities are expressed in the same units. Each unit of flow crossing an arc consumes one unit of its capacity.

A formulation for the integer MFP can be obtained using decision variables that represent the flows in all arcs for all commodities, $x_{ij}^k$, $\forall ij \in A, \ \forall k \in K$. The arc formulation is as follows:

$$Min \sum_{k \in K} \sum_{ij \in A} c_{ij}^k x_{ij}^k \qquad (AFI)$$

*subject to:*

$$\sum_{j:ij \in A} x_{ij}^k - \sum_{j:ji \in A} x_{ji}^k = b_i^k, \ \forall i \in N, \ \forall k \in K \qquad (3.1)$$

$$\sum_{k \in K} x_{ij}^k \leq u_{ij}, \ \forall ij \in A \qquad (3.2)$$

$$x_{ij}^k \geq 0 \text{ and integer}, \ \forall ij \in A, \ \forall k \in K.$$

Constraints *(3.1)* are flow conservation constraints. They state that, for each commodity, the difference between the flow that enters a node and the flow that leaves that node is equal to the supply/demand of that node. Constraints *(3.2)* are capacity constraints. They state that the total flow on each arc must be less than or equal to its capacity.

The integer MFP can be seen as an extension of the minimum cost flow problem: if we neglect the capacities of the arcs, we obtain a set of independent minimum cost flow problems, one for each commodity. However, the optimal solution of the linear relaxation of the integer MFP is not necessarily integer. That marks a clear difference between the two problems. There are several polynomial algorithms available to obtain an integer optimal solution to the minimum cost flow problem, but the integer MFP is NP-hard (Garey and Johnson, 1979).

### 3.2.2 Tree formulations

Taking the arc formulation, *(AFI)*, as the original formulation in a DWD, and defining the subproblem with the flow conservation constraints *(3.1)*, we obtain a minimum cost flow subproblem for each commodity. By denoting the set of extreme points of the subproblem of commodity $k$ by $T^k$ and the flow on the arc $ij$ in the $t$-th solution of subproblem $k$ by $z_{ij}^{tk}$, the master problem is

$$Min \sum_{k \in K} \sum_{t \in T^k} \left( \sum_{ij \in A} c_{ij}^k z_{ij}^{tk} \right) \lambda^{tk} \qquad (TFI)$$

*subject to:*

$$\sum_{k \in K} \sum_{t \in T^k} z_{ij}^{tk} \lambda^{tk} \leq u_{ij}, \ \forall ij \in A$$

$$\sum_{t \in T^k} \lambda^{tk} = 1, \ \forall k \in K$$

$$\sum_{t \in T^k} z_{ij}^{tk} \lambda^{tk} \text{ integer}, \ \forall k \in K, \ \forall ij \in A \qquad (3.3)$$

$$\lambda^{tk} \geq 0, \ \forall k \in K, \ \forall t \in T^k,$$

where the decision variables $\lambda^{tk}$ are the weights of the $t$-th (minimum cost flow) solution of subproblem $k$. The relation between the original and weight variables can be seen through constraints *(3.3)* that state that the original variables $x_{ij}^k, \ \forall ij \in A, \ \forall k \in K$, must take integer values.

When there is only one origin and several destinations for each commodity, or only one destination and several origins for each commodity, the subproblems are shortest path tree problems. When there is only one origin and one destination for each commodity, the subproblems are shortest path problems. In the latter case, the formulation is the same as the path formulation presented next.

Note that every solution of a minimum cost multicommodity flow can be expressed as a set of flows on paths and cycles (for example, (Ahuja et al., 1993)) and that, in an optimal solution, since we assumed $c_{ij}^k \geq 0, \ \forall ij \in A, \ \forall k \in K,$ all the flows on cycles will have zero value. Furthermore, all paths with positive flow have their origin in one supply node and their destination in one demand node.

Being so, model *(TFI)* can be further decomposed by considering the subproblem of each commodity as a set of shortest path problems, one for each origin-destination pair of that commodity. Following this approach, we obtain what we call a path formulation.

### 3.2.3 Path formulations

We now introduce some new notation. We denote the set of all simple paths between all origin-destination pairs of commodity $k$ by $P^k$, and the set of all origins and destinations of commodity $k$ by $Q^k$. If arc $ij$ belongs to path $p$ of commodity $k$, then $y_{ij}^{pk}$ equals *1*, and *0*, otherwise. If node $i$ is an origin of path $p$ of commodity $k$, then $\delta_i^{pk}$ equals *1*; if $i$ is a destination, then $\delta_i^{pk}$ equals *–1*; otherwise, $\delta_i^{pk}$ equals *0*. The unit flow cost of path $p$ of commodity $k$, is represented as $c^{pk} = \sum_{ij \in A} y_{ij}^{pk} c_{ij}^k, \ \forall p \in P^k, \ \forall k \in K$.

The path formulation for the integer MFP is

$$Min \sum_{k \in K} \sum_{p \in P^k} c^{pk} \lambda^{pk} \qquad (PFI)$$

*subject to:*

$$\sum_{p \in P^k} \delta_i^{pk} \lambda^{pk} = b_i^k, \ \forall k \in K, \ \forall i \in Q^k \qquad (3.4)$$

$$\sum_{k \in K} \sum_{p \in P^k} y_{ij}^{pk} \lambda^{pk} \leq u_{ij}, \ \forall ij \in A \qquad (3.5)$$

$$\sum_{p \in P^k} y_{ij}^{pk} \lambda^{pk} \text{ integer}, \ \forall k \in K, \ \forall ij \in A \qquad (3.6)$$

$$\lambda^{pk} \geq 0, \ \forall k \in K, \ \forall p \in P^k,$$

where the decision variables $\lambda^{pk}$ are the flows on each simple path $p$ of commodity $k$.

Constraints *(3.4)* force all the units of all commodities to leave the origins and reach the destinations. The other flow conservation constraints are implicitly considered in the decision variables: only paths linking origin-destination pairs are considered. Constraints *(3.5)* are the capacity constraints.

The integrality of all the flows is forced by constraints *(3.6)*, since

$$x_{ij}^{k} = \sum_{p \in P^k} y_{ij}^{pk} \lambda^{pk}, \ \forall k \in K, \ \forall ij \in A.$$

The integrality constraints could also be imposed directly on the decision variables,

$$\lambda^{pk} \text{ integer}, \ \forall k \in K, \ \forall p \in P^k.$$

This issue will be discussed in more detail in subsection 3.3.2.

When compared with the arc formulation, path formulations have, in general, a smaller number of constraints ( $\sum_{k \in K} q^k + m$ opposed to $hn+m$, where $q^k$ is the number of origins plus the number of destinations of commodity $k$) and a much larger number (exponential with respect to the dimension of the network) number of variables. However, in a basic solution of the path formulation, at most, $\sum_{k \in K} q^k + m$ variables have a positive value. The advantage of using a decomposition approach is clear: as long as an efficient subproblem solver is available (as is the case in the path formulations) we may expect to have much smaller problems to solve, which may be relevant for efficiency. Furthermore, less computational memory is required.

Besides their potential efficiency, another advantage of path formulations can be relevant: modelling a MFP on paths allows the easy consideration of issues that may have an important practical meaning. We give two examples. In telecommunication routing models, a commodity is associated with the traffic between a given origin and destination, and the ability to model delay or survivability constraints is relevant. Those issues may be taken into account in a path formulation by using only paths with a limited number of nodes/arcs. The implication in terms of the column generation method amounts to slightly modifying the subproblem, to generate only paths with the desired characteristics (Holmberg and Yuan, 2001). Another example, taken from a stochastic MFP treated in (Soroush and Mirchandani, 1990), is the situation where the (expected) costs are associated with paths and not with arcs.

When compared to tree formulations, as first noted in (Jones et al., 1993), path formulations tend to be more efficient.

We note that when all commodities have only one origin and one destination, tree and path formulations are equivalent.

### 3.2.4 Review of solution methods

In this subsection we review some of the major approaches that have been proposed to solve the linear MFP and variants.

Specialised simplex methods (for example, (Kennington and Helgason, 1980)) work on the arc formulation, and partitioning the basis in network and non-network parts. This partition allows performing simplex iterations in a specialised way, using a working basis much smaller than the (original full) basis. Recent work on this approach is presented in (Castro and Nabona, 1996; Chardaire and Lisser, 2002b; Detlefsen and Wallace, 2002). Specialised interior point methods, also based on the arc formulation, have also been developed (Castro, 2000; Chardaire and Lisser, 2002b).

In a resource decomposition method, the node arc formulation is decomposed splitting the available capacities among the commodities. A master problem is responsible for specifying the available capacity for each arc and for each commodity, and a set of subproblems (minimum cost flow subproblems with upper bounds corresponding to the available capacities on the arcs), one for each commodity, is considered. This approach amounts to minimising a piecewise linear function whose value is defined by the subproblem. Methods for solving that type of problems (such as subgradient or cutting planes) in the context of this decomposition approach for the linear MFP are described in (Kennington and Shalaby, 1977; Assad, 1978; Kennington, 1978; Ahuja et al., 1993).

The tree and path formulations can be obtained by applying Lagrangean relaxation (and dualising) (Held and Karp, 1970; Held and Karp, 1971; Lemaréchal, 2003) or DWD to the arc formulation. After the reformulation, a (restricted) master problem is responsible for setting the prices of the capacities and there is a set of subproblems (minimum cost flow problems with the costs modified by the prices of the capacities), one for each commodity. Several methods to implement this general approach to the linear MFP, usually referred to as price decomposition methods, have been described in the literature: subgradient (for example, (Saviozzi, 1986)), column generation (for example, (Tomlin, 1966; Jones et al., 1993)), bundle (Frangioni and Gallo, 1999) and analytic center cutting plane (Goffin et al., 1996). A related price decomposition approach, which uses quadratic penalisations in a Lagrangean relaxation context, is given in (Larsson and Yuan, 2004).

A dual ascent heuristic (in a price decomposition context – path formulation) (Barnhart, 1993), a primal-dual heuristic (arc formulation) (Barnhart and Sheffi, 1993) and a scaling algorithm (Schneur and Orlin, 1998) have also been devised for the linear MFP.

Combinations of (some of) the above methods have also been described. In (Farvolden et al., 1993) a simplex specialisation is applied in the path formulation. In (Barnhart et al., 1995) a different formulation (based on representing the flow of each commodity on a key path and cycles) is also used to combine a simplex specialisation with a price directive approach. In both cases, the master problem is not reoptimised but a simplex iteration on a partitioned basis is performed. In (Mamer and McBride, 2000; McBride and Mamer, 2001) a simplex specialisation in the arc formulation is used and the pricing of the non-basic variables is performed by solving path subproblems.

In (McBride and Mamer, 1997; McBride, 1998) a simplex specialisation and a resource directive decomposition heuristic are combined.

Parallel implementations of some of the solution methods mentioned above have also been described, as in (Shetty and Muthukrishnan, 1990) (resource directive), (Pinar and Zenios, 1994) (price directive based on a linear-quadratic penalisations), (Cappanera and Frangioni, 2003) (bundle), and (Castro and Frangioni, 2000) (specialised interior-point).

Examples of approximation algorithms can be found in (Goldberg et al., 1998; Fleischer, 2000).

We now refer to some comparisons of the different implementations described in the literature.

In (Ali et al., 1980), price decomposition (column generation), specialised simplex and resource decomposition (solved by the subgradient method) are compared. The first two approaches spend comparable computational times in obtaining optimal solutions. The third one is faster, but shows (for some instances) convergence difficulties.

A more recent computational comparison is given in (Frangioni and Gallo, 1999): a bundle method is compared with a primal partitioning code (PPRN 1.0 (Castro and Nabona, 1994)) and two general purpose solvers (Cplex 3.0 and LOQO 2.21). The bundle code proved to be the most efficient for several sets of instances, the difference being very meaningful for instances with a large number of commodities.

In (Larsson and Yuan, 2004) some of those methods (namely, bundle, PPRN 1.0, and Cplex 5.0) are compared with a column generation implementation and with the augmented Lagrangean described in the paper. This last method resulted, by far, in the best computational times, obtaining approximate solutions of very good quality (relative duality gap frequently less than 0.1%). From the other methods, column generation provided the best computational times, solving all the instances in reasonable times.

In (Chardaire and Lisser, 2002b), computational tests also showed that the column generation method is more efficient, when compared to Cplex 4.0, specialised simplex and interior point methods presented in the paper, as well as an analytic center cutting plane method.

So far, in this subsection, we only provided references to the *linear* MFP. References to

the *integer* MFP are rare. The heuristic procedure based on a resource decomposition and parametric analysis given in (Aggarwal et al., 1995) is an exception.

## 3.3 Branch-and-Price for the Integer MFP

### 3.3.1 Solving the linear relaxation

The application of column generation for the linear relaxation of the integer MFP where each commodity only has one origin and one destination is described in detail, for example, in (Ahuja et al., 1993). The case where commodities have one origin (destination) and several destinations (origins) can be easily transformed into the previous case by disaggregating the commodities in the model (as shown in (Jones et al., 1993)). Our column generation procedure, also based on a path formulation, is for the general case: each commodity may have several origins and several destinations. The formulation that we use is the path formulation given in subsection 3.2.3 (page 63).

The column generation methodology is based on implicitly considering a large number of variables. In each iteration, a restricted master problem (RMP) and a subproblem are solved. The RMP is initialised with a restricted number of variables. After its optimisation, the values of the dual variables are transferred to the subproblem, which allows pricing the variables that are not present in the RMP. If there are attractive variables, one, or more, columns are inserted in the RMP, this problem is reoptimised, and the iterative process goes on. Otherwise, the optimal solution to the RMP is a provable optimal solution to the original problem.

Some practical questions arise when implementing a column generation scheme, such as how to construct the first RMP and what to do with the columns that are nonbasic after the RMP optimisation. We did some computational tests in an attempt to answer these questions, and will discuss them later.

In the resolution of the linear relaxation of the integer MFP using column generation, the RMP is initialised with a reduced number of paths. After optimising the RMP, we evaluate the attractiveness of the paths that are not present in the RMP by solving a subproblem that uses the values of the dual variables. The subproblem consists in determining the shortest path between all the origin-destination pairs of all the commodities in a network with modified costs, as follows.

Representing the (nonnegative) dual variable associated with the capacity constraint of the arc $ij$ as $w_{ij,}$ and the (unrestricted in sign) dual variable associated with the flow conservation constraint of node $i$ for commodity $k$ as $\pi_i^k$, the reduced cost of a path is given by

$$\bar{c}^{pk} = \sum_{ij \in A} y_{ij}^{pk}(w_{ij} + c_{ij}^{k}) - \pi_o^k + \pi_d^k, \qquad (3.7)$$

where the indices $o$ and $d$ represent the origin and destination nodes, respectively, of path $p$. All paths that belong to the RMP have nonnegative reduced costs, as follows from the linear programming optimality conditions. The optimal solution of the RMP will not be the optimal solution for the linear relaxation of the integer MFP, if there is a path not present in the RMP with negative reduced cost. In equation *(3.7)*, for a given commodity $k$, the quantity $-\pi_o^k + \pi_d^k$ is a constant for all the paths that have the same origin and the same destination. Each one of the remaining terms is associated with an arc, being constant for all the paths of a given commodity that include the arc.

For a commodity $k$, the path with the smallest reduced cost between an origin $o$ and a destination $d$ is the shortest path between $o$ and $d$ in a network where the costs of the arcs are given by $w_{ij} + c_{ij}^k, \ \forall ij \in A$.

This path – denoted as $p$ – is attractive if

$$\sum_{ij \in A} y_{ij}^{pk}(w_{ij} + c_{ij}^k) < \pi_o^k - \pi_d^k.$$

Otherwise, it is guaranteed that there are no attractive paths for commodity $k$ for the origin-destination pair *o-d*. Determining the most attractive path for each commodity corresponds to solving a shortest path problem for each origin-destination pair. We note that the network may have cycles, but since $c_{ij}^k \geq 0$ and $w_{ij} \geq 0, \ \forall ij \in A$, their cost is always nonnegative, and thus each shortest path problem has a finite solution and may be solved by Dijkstra's algorithm (for example, (Gallo and Pallottino, 1988)).

### 3.3.2 Branching rules

In order to obtain an optimal solution to the integral MFP, we combine column generation with branch-and-bound. The main issue is the branching scheme. Below, we first discuss a branching rule based on the path variables, and then we propose one that is based on the arc variables.

As noted in subsection 3.2.3 (page 63), the relation between arc flows and path flows is given by

$$x_{ij}^k = \sum_{p \in P^k} y_{ij}^{pk} \lambda^{pk}, \ \forall k \in K, \ \forall ij \in A; \qquad (3.8)$$

thus, forcing the integrality of the flows in arcs is the same as forcing the integrality of the flows in paths. Being so, integrality constraints may be imposed directly on the path variables,

$$\lambda^{pk} \text{ integer}, \ \forall k \in K, \ \forall p \in P^k,$$

which leads to branching constraints of the type

$$\lambda^{pk} \leq \left\lfloor \bar{\lambda}^{pk} \right\rfloor \text{ and } \lambda^{pk} \geq \left\lfloor \bar{\lambda}^{pk} \right\rfloor + 1,$$

where $\bar{\lambda}^{pk}$ denotes the current (fractional) flow of the path *p* of commodity *k*.

With these branching constraints in the RMP of a node of the search tree, the subproblem must take into account their dual variables. The issue here is that the dual variables of the branching constraints are associated with path variables, and the decision variables of the shortest path (sub)problem are related with flows on arcs. A way of overcoming this difficulty is to neglect the duals of the branching constraints and, if a path that is already being considered in the RMP is (re)generated by the subproblem, then the second best path is sought. In a node of the search tree with *k* branching constraints, this approach may lead to the *k*-shortest paths problem, making the subproblem much more difficult to solve than the one of the root node.

Branching in the arc variables does not pose the regeneration difficulty. Given the relation between the arc and the path variables, expressed in *(3.8)*, it is always possible to obtain the flow in an arc based on the flows in paths.

Branching constraints of the type

$$\sum_{p \in P^k} y_{ij}^{pk} \lambda^{pk} \leq \left\lfloor \bar{x}_{ij}^{k} \right\rfloor \text{ and } \sum_{p \in P^k} y_{ij}^{pk} \lambda^{pk} \geq \left\lfloor \bar{x}_{ij}^{k} \right\rfloor + 1,$$

where $\bar{x}_{ij}^{k}$ denotes the current (fractional) flow of commodity *k* in the arc *ij*, overcome the regeneration difficulties in the subproblem. Since these constraints are associated with flows in arcs, their duals are considered as the duals of the capacity constraints in the coefficients of the objective function.

Using this branching rule and representing $U^s$ as the set of branching constraints of type *"≤"*, indexed by *u*, and $L^s$ as the set of branching constraints of type *"≥"*, indexed by *l*, the RMP of a node *s* of the search tree is

$$Min \sum_{k \in K} \sum_{p \in \bar{P}^k} c^{pk} \lambda^{pk}$$

*subject to:*

$$\sum_{p \in \bar{P}^k} \delta_i^{pk} \lambda^{pk} = b_i^k, \ \forall k \in K, \ \forall i \in Q^k$$

$$\sum_{k \in K} \sum_{p \in \bar{P}^k} y_{ij}^{pk} \lambda^{pk} \leq u_{ij}, \ \forall ij \in A$$

$$\sum_{p \in \bar{P}^k} y_{ij}^{pk} \lambda^{pk} \leq \left\lfloor x_{ij}^{ku} \right\rfloor, \ \forall u \in U^s \qquad (3.9)$$

$$\sum_{p \in \bar{P}^k} y_{ij}^{pk} \lambda^{pk} \geq \left\lceil x_{ij}^{kl} \right\rceil, \ \forall l \in L^s \qquad (3.10)$$

$$\lambda^{pk} \geq 0, \ \forall k \in K, \ \forall p \in \bar{P}^k,$$

where $\overline{P}^k$ is the set of paths of commodity $k$ that belong to the RMP, $\overline{P}^k \subseteq P^k$, $\forall k \in K$.

The reduced cost of a path $p$, with origin $o$ and destination $d$, of a commodity $k$ is now:

$$\bar{c}^{pk} = \sum_{ij \in A} y_{ij}^{pk} (w_{ij} + \sum_{u \in U^s} w_{ij}^{ku} - \sum_{l \in L^s} w_{ij}^{kl} + c_{ij}^{k}) - \pi_o^k + \pi_d^k,$$

where $w_{ij}^{ku}$ and $w_{ij}^{kl}$ are the dual variables associated with the branching constraints *(3.9)* and *(3.10)*, respectively. Note that these variables are associated with arcs, and thus they can be included in the modified costs of the subproblem.

Therefore, the subproblem continues to be a shortest path problem between all the origin-destination pairs of each commodity in a network with modified costs. The modified cost of an arc $ij$, for a commodity $k$, is given by:

$$w_{ij} + \sum_{u \in U^s} w_{ij}^{ku} - \sum_{l \in L^s} w_{ij}^{kl} + c_{ij}^{k}.$$

In the linear relaxation, there was the guarantee that all costs in the shortest path (sub)problems were positive. With the inclusion of branching constraints of type *"≥"* that guarantee does not hold anymore. This implies that the solution of the shortest path (sub)problem may be unbounded: a cycle with a negative cost may exist in its network. This issue could be overcome by considering only elementary paths. However, that approach is not promising, given that the shortest elementary path problem in a network with negative cost cycles is a NP-hard problem (Garey and Johnson, 1979).

In the following subsection we manage to overcome this issue by explicitly considering cycle variables in the path formulation.

### 3.3.3 Dealing with negative cost cycles

The existence of a negative cost cycle in the subproblem of a commodity $k$ is due the presence of branching constraints of type "≥" in the RMP. These constraints may force the existence of a positive flow in a cycle, as illustrated in Figure 3.1, where branching constraints of type "≥" on variables $x_{12}^k$, $x_{23}^k$ and $x_{31}^k$, for some commodity $k$, were imposed. When solving the subproblem of commodity $k$, the network may have negative cost cycles, because the sum of the modified cost of the arcs *12*, *23* and *31* may be negative.

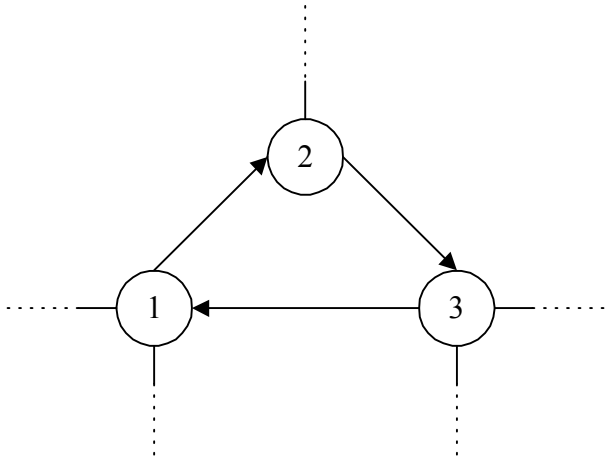



Figure 3.1 Illustration of a cycle with positive flow forced by branching constraints.

Our approach consists in considering a formulation with variables associated with flows in cycles. The problem to solve on a node of the search tree is now

$$Min \sum_{k\in K}\sum_{p\in P^k} c^{pk}\lambda^{pk} + \sum_{k\in K}\sum_{c\in C^k} c^{ck}\mu^{ck}$$

*subject to:*

$$\sum_{p\in P^k} \delta_i^{pk}\,\lambda^{pk} = b_i^k\,,\ \forall k\in K,\ \forall i\in Q^k$$

$$\sum_{k\in K}\sum_{p\in P^k} y_{ij}^{pk}\,\lambda^{pk} + \sum_{k\in K}\sum_{c\in C^k} y_{ij}^{ck}\,\mu^{ck} \le u_{ij}\,,\ \forall ij\in A$$

$$\sum_{p\in P^k} y_{ij}^{pk}\,\lambda^{pk} + \sum_{p\in P^k} y_{ij}^{ck}\,\mu^{ck} \le \lfloor x_{ij}^{ku} \rfloor,\ \forall u\in U^s$$

$$\sum_{p\in P^k} y_{ij}^{pk}\,\lambda^{pk} + \sum_{p\in P^k} y_{ij}^{ck}\,\mu^{ck} \ge \lceil x_{ij}^{kl} \rceil,\ \forall l\in L^s$$

$$\lambda^{pk} \ge 0,\ \forall k\in K,\ \forall p\in P^k$$

$$\mu^{ck} \ge 0,\ \forall k\in K,\ \forall c\in C^k,$$

where the set of all cycles of commodity $k$ is denoted as $C^k$, indexed by $c$; if arc $ij$ belongs to cycle $c$ of commodity $k$, then $y_{ij}^{ck}$ equals *1*, and *0*, otherwise; the unit flow cost of cycle $c$ of commodity $k$, is represented as $c^{ck} = \sum_{ij\in A} y_{ij}^{ck}\,c_{ij}^k,\ \forall c\in C^k,\ \forall k\in K$; the $\mu^{ck}$ variables are associated with the flow in each cycle of each commodity. Note that the variables associated with the cycles do not appear in the flow conservation constraints, but do appear in the capacity and branching constraints.

The reduced cost of a variable associated with a cycle $c$ of a commodity $k$ is

$$\bar{c}^{ck} = \sum_{ij\in A} y_{ij}^{ck}\,(w_{ij} + \sum_{u\in U^s} w_{ij}^{ku} - \sum_{l\in L^s} w_{ij}^{kl} + c_{ij}^k).$$

The reduced cost of a cycle is equal to the sum of the modified costs of the arcs that form that cycle. The existence of a negative cost cycle in the subproblem corresponds to the existence of an attractive cycle.

In the nodes of the search tree, other than the root, the subproblems must be solved using an algorithm that identifies negative cost cycles, such as the labelling correcting algorithms (for example, (Gallo and Pallottino, 1988)). Each subproblem of a node of the search tree can return to the RMP a column that corresponds either to a cycle, if one is detected, or to a path, if no negative cycles are detected.

It is relevant to note that, in an optimal integer solution, the variables associated with cycles have, necessarily, a null value.

The branch-and-price algorithm presented here for the integer MFP has finite convergence, since the number of paths and cycles of a network is finite (although exponentially large with respect to the size of the network). The same can be said about the number of branching constraints (each branch is defined by one arc and one commodity, and thus its number is finite). So, there is the guarantee that the method obtains an optimal solution in a finite number of steps, which, in the worst case, may be exponential (as in general branch-and-bound algorithms).

## 3.4 Implementation Issues and Computational Results

### 3.4.1 Objectives of the computational tests

We implemented the proposed method and did some computational experiments with the following objectives:

– to evaluate comparatively different alternatives for the branch-and-price algorithm, namely the method to obtain the first RMP, the criterion to remove columns and the algorithm to solve the RMPs;

– to test experimentally the sensitivity of the method to the type of subproblem instances;

– to compare the efficiency of the proposed method with that of a general-purpose linear and integer programming solver (Cplex 6.6 (ILOG, 1999)).

In the comparison of our branch-and-price algorithm with the software package for solving general integer programs, we remark the considerable evolution, in recent years, of mathematical programming software, which has incorporated both sophisticated software implementation techniques and linear/integer programming theoretical concepts, such as the use of heuristics, strong branching, node pre-solve and cutting planes in the nodes of the branch-and-bound tree (Bixby et al., 2000).

We performed two types of tests. In the preliminary tests, we ran the BP program (the implementation of our algorithm) as well as the Cplex 6.6 (each with different alternatives) to solve the linear relaxation of the integer MFP for a small set of instances. In the comparative tests, we solved all the instances with the alternatives that provided better results in the first group of tests.

### 3.4.2 Test instances

We performed computational experiments with five sets of instances, four of them taken from (Frangioni, 2005) and the fifth generated by the random generator *Mnetgen* available at the same site. We used the C++ service class *graph*, also available at the same site, for easiness of conversion between different formats. We now briefly describe the instances tested (for a more detailed description, as well as for their origin, we refer the reader to the reference mentioned above).

One instance of a MFP can be characterised by the problems of each commodity when relaxing the capacity constraints: in problems in which each commodity is associated with an origin-destination pair, if we relax the capacity constraints, we will get a shortest path problem for each commodity. In the same way, we can get shortest path tree subproblems (one origin and several destinations, or one destination and several origins) or minimum cost flow subproblems (several origins and several destinations). When describing the types of instances, we refer to those subproblems, noting that the characterisation used bears no relation to the solution method, but rather to the type of instance.

***Aertranspo***

For each of the eight instances of this set, each commodity has just one origin and several destination nodes (so the subproblem is a shortest path tree). The numbers of nodes, arcs and commodities for each instance are given in the Table 3.1.

| *Instance* | *jl23* | *jl049* | *jl141* | *jl147* | *jl158* | *jl188* | *jl207* | *jl209* |
|---|---|---|---|---|---|---|---|---|
| *n* | 23 | 49 | 141 | 147 | 158 | 188 | 207 | 209 |
| *m* | 71 | 137 | 449 | 520 | 477 | 673 | 726 | 765 |
| *h* | 18 | 40 | 132 | 140 | 138 | 166 | 189 | 194 |

Table 3.1 Dimensions of the *Aertranspo* instances.

***Canad***

The *Canad* set of instances consists in three subsets: *Bipart*, *Mulgen I* and *Mulgen II*.

The *Bipart* instances are defined over a bipartite network and the subproblems are

minimum cost flow problems without upper bounds for each commodity (more precisely transportation problems, since the network is bipartite).

The *Mulgen I* instances have shortest path subproblems and the *Mulgen II* instances have minimum cost flow subproblems (over generic networks).

The sizes of all the *Canad* instances are given in Table 3.2, Table 3.3, and Table 3.4.

| *Instance* | *p01-p04* | *p05-p08* | *p09-p12* | *p13-p16* | *p17-p20* | *p21-p24* | *p25-p28* | *p29-p32* |
|---|---|---|---|---|---|---|---|---|
| *n* | 50 | 50 | 50 | 50 | 100 | 100 | 100 | 100 |
| *m* | 400 | 400 | 625 | 625 | 1600 | 1600 | 2500 | 2500 |
| *h* | 10 | 100 | 10 | 100 | 10 | 100 | 10 | 100 |

Table 3.2 Dimensions of the *Canad* instances (subset *Bipart*).

| *Instance* | *p33-p36* | *p37-p40* | *p41-p44* | *p45-p48* | *p49-p52* | *p53-p56* | *p57-p60* | *p61-p64* |
|---|---|---|---|---|---|---|---|---|
| *n* | 20 | 20 | 20 | 20 | 30 | 30 | 30 | 30 |
| *m* | 230 | 229 | 289 | 287 | 517 | 519 | 669 | 688 |
| *h* | 40 | 200 | 40 | 200 | 100 | 400 | 100 | 400 |

Table 3.3 Dimensions of the *Canad* instances (subset *Mulgen I*).

| *Instance* | *p65-p68* | *p69-p72* | *p73-p76* | *p77-p80* | *p81-p84* | *p85-p88* | *p89-p92* | *p93-p96* |
|---|---|---|---|---|---|---|---|---|
| *n* | 20 | 20 | 20 | 20 | 30 | 30 | 30 | 30 |
| *m* | 230 | 229 | 289 | 287 | 517 | 519 | 669 | 688 |
| *h* | 40 | 200 | 40 | 200 | 100 | 400 | 100 | 400 |

Table 3.4 Dimensions of the *Canad* instances (subset *Mulgen II*).

We note that these instances come from a fixed charge multicommodity flow problem, and that the only difference between the instances that form each consecutive pair is the fixed charge cost. As we do not consider the fixed cost, we only tested half of the original instances.

We also neglected the individual upper bounds on commodities that these instances originally had.

***Mnetgen***

We generated three sets of instances with the generator *Mnetgen*. For the first one we generated instances in which the subproblem is a minimum cost flow problem for each commodity (with an equal number of origins and destinations). For the second one the subproblem is a shortest path tree for each commodity (in which half of the nodes are

transhipment nodes). For the third one the subproblem is a shortest path for each commodity.

All the instances have 256 nodes. For the first set we generated four subsets of 12 instances (4, 8, 16, and 32 commodities) and for the second and third sets we generated seven subsets of 12 instances with 4, 8, 16, 32, 64, 128, and 256 commodities.

In Table 3.5 we present some of the input parameters for each of the subsets.

| *Instances* | *1,2,3* | *4,5,6* | *7,8,9* | *10,11,12* |
|---|---|---|---|---|
| *Density* | low | low | high | high |
| *Capacitated arcs (%)* | 40 | 80 | 40 | 80 |
| *Arcs with maximum cost (%)* | 10 | 30 | 10 | 30 |

Table 3.5 Most relevant input parameters for the *Mnetgen* instances.

We neglected the individual upper bounds on commodities that the *Mnetgen* generator specifies.

The 1st, 2nd, 3rd, 6th, 7th, and 8th instances of each subset are expected to be easier to solve, given that they have a lower number of capacitated arcs and a higher number of arcs with maximum cost. Besides the internet site where this instance generator was obtained, the reader can find a related explanation in (Klingman et al., 1974), where the *Netgen* generator (a single commodity instance generator that is the base of *Mnetgen*) is presented.

To identify each instance, the number of commodities precedes its number. If the name of the instance does not contain a letter it refers to an instance of the first set (minimum cost flow subproblems); if it contains an ‘o’ it is an instance of the second set (shortest path tree subproblems); and if it contains a ‘c’ it is an instance of the third set (shortest path subproblems).

***PDS***

This is probably the most exhaustively tested group of instances of MFPs. It has been used by several researchers to test specialised approaches for linear MFPs and also as a typical large linear program (as in (Carolan et al., 1990) and (Bixby et al., 2000)).

We tested the *PDS* instances given on Table 3.6 and Table 3.7. Each commodity is associated with an origin-destination pair, so the subproblem is a shortest path. Each instance has 11 commodities.

| *Instance* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *n* | 126 | 252 | 390 | 541 | 686 | 835 | 971 | 1104 | 1253 | 1399 | 1541 | 1692 | 1837 | 1981 |
| *m* | 339 | 685 | 1117 | 1654 | 2149 | 2605 | 2989 | 3348 | 3880 | 4433 | 4945 | 5527 | 6095 | 6636 |

Table 3.6 Dimensions of the first 14 *PDS* instances.

| *Instance* | *15* | *18* | *20* | *21* | *24* | *27* | *30* | *33* | *36* | *40* | *80* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *n* | 2125 | 2558 | 2857 | 2996 | 3419 | 3823 | 4223 | 4643 | 5081 | 5652 | 10989 |
| *m* | 7202 | 8925 | 1115 | 10630 | 12199 | 13662 | 15125 | 16721 | 18449 | 20697 | 40258 |

Table 3.7 Dimensions of the 11 bigger *PDS* instances that were tested.

We neglected the individual upper bounds on commodities that these instances originally had.

***Planar***

The *planar* instances are defined over a planar graph. The subproblem is a shortest path problem for each commodity. For each arc, all the commodities have the same cost. We tested the instances given on Table 3.8.

| *Instance* | *planar30* | *planar50* | *planar80* | *planar100* | *planar150* |
|---|---|---|---|---|---|
| *n* | 30 | 50 | 80 | 100 | 150 |
| *m* | 150 | 250 | 440 | 532 | 850 |
| *h* | 92 | 267 | 543 | 1085 | 2239 |

Table 3.8 Dimensions of the *Planar* instances.

### 3.4.3 Implementation issues and preliminary tests

We implemented the proposed method in C++ using the development environment Microsoft Visual Studio 6.0. We used Cplex 6.6 callable library to solve linear programs and LEDA 4.1 (Mehlhorn and Näher, 1999) to keep the network topology and data, and to solve shortest path problems. We refer to this computer program as BP.

The obvious advantage of using a library of classes, as LEDA, is that it becomes easier to code an algorithm. Also, we can expect some algorithms (as the shortest path ones) to run more quickly. The main disadvantage is that the usage of memory becomes much higher, since we have to keep two heavy (and partially duplicated) data structures: the network data (in LEDA) and the linear programming data (in Cplex). All in all, since LEDA makes it possible to extend our code to implement other ideas regarding the same problem, or to extend the present

approach to other MFPs easily, we decided to sacrifice the computational tests on larger instances.

We also implemented a program based on Cplex 6.6 callable library to solve the arc formulation of the integer MFP.

The reported results were obtained on a personal computer equipped with a Pentium III, 733 MHz processor, 256 Mb of RAM, running Windows ME. All the times are expressed in seconds and exclude input and output operations. In the tables, the best execution times are presented in bold.

For the preliminary computational tests we selected one instance of each set. The chosen instances were: *jl209* (*Aertranspo*), *p31* (*Canad − Bipart*), *p63* (*Canad − Mulgen I*), *p95* (*Canad − Mulgen II*), all the 12th *Mnetgen* instances, *PDS20*, and *Planar100*.

**Method to obtain the first RMP**

We compared three different ways of generating the set of paths to be included in the first RMP. The common aspect of the three alternatives is that we solve a minimum cost flow problem for each commodity and then convert the arc flows to path flows (with the algorithm presented in (Ahuja et al., 1993)). This set of paths corresponds to the columns of the first RMP.

For the first alternative, the minimum cost flow problem (of each commodity) has upper bounds equal to the capacities of the arcs. An artificial variable, associated to the arcs, is included in the RMP to guarantee a feasible solution. In this way an optimal solution to the first RMP can violate the capacity constraints but never the flow conservation constraints. We refer to this alternative as capacity relaxation.

For the second and third alternatives, we start by setting the upper bounds equal to the capacities of the arcs, but as we solve minimum cost flow problems, we reduce the capacities available, that is, the upper bounds, for the next commodities by the amount of flow that already exists on the arcs.

For the second alternative, if the problem of a commodity is unfeasible, we solve it again with the upper bounds equal to the capacities. As in the first alternative, an artificial variable is inserted in the RMP, and an optimal solution to the first RMP can violate the capacity constraints but never the flow conservation constraints. We refer to this alternative as capacity relaxation with upper bounds.

For the third alternative, we add a super-origin and a super-destination to each minimum cost flow problem. A solution can be divided in two types of flow: the one that enters the original network passing through some origin and arriving at the some destination, and the one that cannot traverse the original network because of tight upper bound constraints, thus traversing the arc that links the super-origin to the super-destination. When the conversion procedure of arc flows to path flows is applied, the existence of this second type of flow will be

reflected in the fact that some of the origins (destinations) cannot send (receive) their supply (demand). An artificial variable is then associated with each of the flow conservation constraints of those origins (destinations). In this alternative, an optimal solution to the first RMP can violate the flow conservation constraints but never the capacity constraints. We refer to this alternative as flow conservation relaxation.

In these preliminary tests, columns with a positive reduced cost were removed at every iteration and the primal simplex algorithm was used to optimise the RMPs.

In Table 3.9 results are given for the methods to obtain the first RMP.

We note that the second and third methods can obtain the same solution if a feasible solution is obtained.

Besides the (obvious) suggestion that a better (less unfeasible in this case) initial solution is preferable, these tests also suggest that the effect of inserting poor quality columns (in the sense that they will not be positive in an optimal solution) can be very significant. It is better to have an initial solution with part of the supply not being delivered rather than having it delivered by a set of paths that exceed the capacity. It is not worth inserting columns based on dual values that are possibly far away from an optimal value.

These results can also be explained in the context of Lagrangean duality. Column generation can be viewed as a method to, iteratively, approximate a piecewise linear function (the Lagrangean dual). Roughly speaking, when that approximation around the optimal solution is good enough, the optimisation of the approximated function has the same result as the optimisation of the original function, in the sense that both functions have the same optimal solution. It seems clear that obtaining a greater accuracy in the approximation for regions far away from the optimal solution is just time-consuming.

For the following tests, the first RMP was obtained by relaxing the flow conservation constraints.

**Removal of columns**

In Table 3.10, we show the results of three strategies related to the removal of columns. The three different alternatives are “never remove columns” (column *Never*), “remove all columns with strictly positive reduced-cost at every iteration” (column *PRC*) and “remove all nonbasic columns at every iteration” (column *NBC*). We used the primal algorithm to optimise RMPs.

We note that the results are different for the last two alternatives because of the degeneracy of the RMP (we defined that a reduced cost is positive if it is greater than $10^{-6}$).

Maintaining only the basic columns in the RMP is clearly worse than the other two alternatives. The comparison of the result of the instance *256-12o*, when columns are not removed, with the one of the *128-12o* instance is surprising, is that an instance with twice the

number of commodities has a smaller solving time, especially when the percentage of saturated arcs in an optimal solution is approximately the same (*7.3* vs. *7.2*). It would be necessary to study in more depth the structure of those instances to explain this result.

| *Instance* | *Capacity relaxation* | *Capacity relaxation with upper bounds* | *Flow conservation relaxation* |
|---|---|---|---|
| *jl209* | 220.0 | 227.1 | **120.6** |
| *p31* | 22.2 | **4.4** | **4.4** |
| *p63* | 3.9 | **1.1** | **1.1** |
| *p95* | 40.6 | 11.7 | **11.7** |
| *4-12* | 57.5 | **8.0** | **8.0** |
| *8-12* | 246.0 | **73.1** | **73.1** |
| *16-12* | 454.2 | **150.5** | **150.5** |
| *32-12* | 823.6 | 710.7 | **404.4** |
| *4-12o* | 7.6 | **0.8** | **0.8** |
| *8-12o* | 15.6 | **1.9** | **1.9** |
| *16-12o* | 28.5 | **9.5** | **9.5** |
| *32-12o* | 90.1 | **12.4** | **12.4** |
| *64-12o* | 345.0 | **43.2** | **43.2** |
| *128-12o* | 1179.9 | **69.0** | **69.0** |
| *256-12o* | 4958.8 | **180.0** | **180.0** |
| *4-12c* | 1.9 | **0.2** | **0.2** |
| *8-12c* | 1.0 | **0.5** | **0.5** |
| *16-12c* | 3.1 | **1.2** | **1.2** |
| *32-12c* | 4.0 | **1.6** | **1.6** |
| *64-12c* | 18.8 | **3.1** | **3.1** |
| *128-12c* | 43.7 | **9.6** | **9.6** |
| *256-12c* | 197.7 | **26.9** | **26.9** |
| *pds20* | 875.9 | **70.8** | **70.8** |
| *planar100* | 403.9 | 79.0 | **56.3** |

Table 3.9 Comparative results of the methods to obtain the first RMP.

In general, on the smaller instances, it is preferable to maintain all the columns in the RMP. As the size of the instances grows larger it is more efficient to remove the columns with positive reduced cost. We can conclude that, while the linear programming solver can efficiently optimise the RMPs, it is preferable not to remove columns. In the view of the discussion of the previous subsection, it is always better to have the best possible approximation

to the dual Lagrangean function that we are optimising. When removing columns we are losing quality in the approximation. Of course, if the linear programming solver cannot manage the size of the RMPs, it is better to have a worse approximation but still optimising it.

We decided to never remove columns in all sets of instances, except *Aertranspo*.

Theoretically, even though it is a remote possibility, removing columns can lead to cycling (see Chapter 2, subsection 2.2.5, page 25). In practice, we never observed that in our algorithm.

**Algorithm to solve the RMPs**

The Cplex algorithms used to solve the RMPs that we tested were: primal (P), dual (D), hybrid primal with preprocessing (HPP), hybrid dual with preprocessing (HDP), hybrid primal without preprocessing (HP) and hybrid dual without preprocessing (HD). The hybrid algorithms use an advanced basis obtained by solving the network type part of the problem. Preprocessing can destroy the network structure, so it is not clear if it should be used with hybrid algorithms. Table 3.11 presents the results obtained.

The primal algorithm is the best alternative for almost all instances. The hybrid approaches are significantly worse. In the remaining tests we used the primal algorithm to solve the RMPs.

**Cplex**

We tested the four basic available alternatives for the optimisation algorithm of the nodes (including the root) of the branch-and-bound(-and-cut) tree when solving the instances with Cplex 6.6. Table 3.12 shows the results obtained. The *Network* column refers to the network algorithm followed by a dual one.

We chose the best algorithm for each set of instances: Network for instances *Aertranspo* and *Canad*, Primal for *PDS* and Dual for the *Planar* instances. For the *Mnetgen* instances the best algorithm is not always the same. We chose the dual algorithm for its robustness.

| *Instance* | *Never* | *PRC* | *NBC* |
|---|---|---|---|
| *jl209* | 420.0 | **120.6** | 185.4 |
| *p31* | **4.4** | 4.8 | 4.8 |
| *p63* | **1.0** | 1.1 | 1.2 |
| *p95* | **10.9** | 11.7 | 40.0 |
| *4-12* | **5.4** | 8.0 | 9.3 |
| *8-12* | **67.6** | 73.1 | 93.1 |
| *16-12* | **136.7** | 150.5 | 176.3 |
| *32-12* | **398.3** | 404.4 | 520.5 |
| *4-12o* | **0.3** | 0.8 | 1.5 |
| *8-12o* | **0.9** | 1.9 | 3.2 |
| *16-12o* | **9.0** | 9.5 | 17.5 |
| *32-12o* | **12.2** | 12.4 | 98.3 |
| *64-12o* | **44.0** | 43.2 | 253.6 |
| *128-12o* | 105.7 | **69.0** | 586.9 |
| *256-12o* | **99.0** | 180.0 | 729.4 |
| *4-12c* | **0.1** | 0.2 | 0.2 |
| *8-12c* | **0.3** | 0.5 | **0.3** |
| *16-12c* | **0.7** | 1.2 | 1.2 |
| *32-12c* | **0.7** | 1.6 | 1.7 |
| *64-12c* | **1.8** | 3.1 | 3.1 |
| *128-12c* | **4.7** | 9.6 | 10.4 |
| *256-12c* | **9.1** | 26.9 | 27.1 |
| *pds20* | **51.1** | 70.8 | 79.4 |
| *planar100* | **39.3** | 56.3 | 12.5 |

Table 3.10 Comparative results for the removal of columns.

| *Instance* | *P* | *D* | *HPP* | *HDP* | *HP* | *HD* |
|---|---|---|---|---|---|---|
| *jl209* | 120.6 | 523.9 | **72.2** | 741.4 | 706.7 | 728.1 |
| *p31* | 4.4 | **4.3** | 21.1 | 23.4 | 22.5 | 34.6 |
| *p63* | **1.0** | 1.2 | 1.5 | 1.3 | 1.6 | 1.3 |
| *p95* | **10.9** | 17.6 | 23.2 | 29.1 | 21.1 | 24.8 |
| *4-12* | **5.4** | 7.7 | 28.0 | 11.0 | 23.9 | 9.1 |
| *8-12* | **67.6** | 115.1 | 613.3 | 147.3 | 651.2 | 159.9 |
| *16-12* | 136.7 | **31.5** | 1160.1 | 1191.9 | 1562.5 | 600.9 |
| *32-12* | **398.3** | 997.6 | 2863.3 | 1414.0 | 4191.4 | 1630.2 |
| *4-12o* | **0.3** | 0.6 | 2.8 | 1.0 | 2.8 | 1.0 |
| *8-12o* | **0.9** | 2.2 | 8.6 | 5.8 | 11.2 | 5.7 |
| *16-12o* | **9.0** | 20.3 | 35.3 | 69.3 | 43.5 | 67.8 |
| *32-12o* | **12.2** | 28.9 | 81.7 | 160.7 | 75.1 | 142.5 |
| *64-12o* | **44.0** | 88.0 | 992.0 | 650.0 | 995.1 | 643.6 |
| *128-12o* | **105.7** | 275.8 | 1783.7 | 1745.8 | 1778.4 | 445.7 |
| *256-12o* | **99.0** | 345.5 | 6251.3 | 12888.4 | 11131.8 | 12956.6 |
| *4-12c* | **0.1** | 0.2 | 0.5 | 0.2 | 0.5 | 0.2 |
| *8-12c* | **0.3** | 0.4 | 1.1 | 0.5 | 1.1 | 0.5 |
| *16-12c* | **0.7** | 1.0 | 2.8 | 1.3 | 2.6 | 1.2 |
| *32-12c* | **0.7** | 1.1 | 1.9 | 1.1 | 1.8 | 1.2 |
| *64-12c* | **1.8** | 2.7 | 3.2 | 2.8 | 3.4 | 4.3 |
| *128-12c* | **4.7** | 6.5 | 10.4 | 7.8 | 10.9 | 10.8 |
| *256-12c* | **9.1** | 15.3 | 43.4 | 29.7 | 43.7 | 30.5 |
| *pds20* | **51.1** | 79.0 | 545.5 | 185.9 | 415.6 | 185.9 |
| *planar100* | **39.3** | 57.7 | 113.6 | 87.3 | 105.6 | 74.0 |

Table 3.11 Alternatives for the algorithm to optimise RMPs.

| *Instance* | *Primal* | *Dual* | *Network* | *Barrier* |
|---|---|---|---|---|
| *jl209* | 1114.7 | 556.4 | **289.6** | 546.7 |
| *p31* | 54.0 | 99.4 | **17.1** | 1739.5 |
| *p63* | 150.4 | 18.8 | **15.2** | 860.5 |
| *p95* | 2638.5 | 32.8 | **28.3** | 912.8 |
| *4-12* | 23.6 | **11.3** | 15.3 | 32.3 |
| *8-12* | 211.5 | **116.8** | 198.0 | 779.0 |
| *16-12* | 541.1 | **53.8** | 75.1 | 559.9 |
| *32-12* | 494.0 | 87.6 | **23.9** | 313.5 |
| *4-12o* | 1.2 | **1.0** | 1.1 | 8.7 |
| *8-12o* | 3.7 | 2.3 | **1.8** | 43.3 |
| *16-12o* | 17.1 | 8.0 | **6.0** | 190.9 |
| *32-12o* | 24.0 | 13.2 | **8.6** | 141.6 |
| *64-12o* | 98.4 | 27.3 | **22.1** | 157.4 |
| *128-12o* | 55.5 | **47.2** | 48.6 | 225.0 |
| *256-12o* | 2082.5 | 285.7 | **275.2** | 1345.2 |
| *4-12c* | 0.8 | **0.4** | **0.4** | 7.9 |
| *8-12c* | 2.6 | **0.9** | 1.1 | 41.9 |
| *16-12c* | 8.7 | **2.9** | 3.5 | 215.1 |
| *32-12c* | 41.9 | **23.0** | 44.0 | 362.3 |
| *64-12c* | 21.4 | **15.5** | 22.3 | 211.6 |
| *128-12c* | **83.1** | 89.2 | 110.3 | 188.3 |
| *256-12c* | **74.7** | 93.8 | 124.4 | 145.1 |
| *pds20* | **33.7** | 82.8 | 49.9 | 924.5 |
| *planar100* | 9129.7 | **1462.1** | 7582.6 | 7810.5 |

Table 3.12 Alternatives for the Cplex algorithm.

### 3.4.4 Comparative computational tests

For testing all instances we used the alternatives fixed in the previous subsection. For traversing the branch-and-price tree we used a depth first strategy. The branching rule defined is to branch on the first fractional arc variable found. More precisely, the flow of each arc for each commodity is determined until a fractional value is obtained, and then branching is performed as proposed in subsection 3.3.2 (page 68).

In all the tables the first column is the number of saturated arcs in an optimal solution to the linear relaxation, which, in general, is a measure of the difficulty of the instance. The column “gap” gives the absolute value of the integrality gap, which is the optimal integer value

minus the optimal value of the linear relaxation. The columns *Time* and *Nodes* present, respectively, the time spent in obtaining the optimal solution and the number of nodes (including the root one) that were optimised with BP and Cplex 6.6.

***Aertranspo***

In Table 3.13 the results for the *Aertranpo* instances are given.

For all the instances the BP program was more efficient than Cplex 6.6. On average, BP is about three times faster than Cplex 6.6. That can be explained by the smaller number of constraints in the master problem of the BP approach, since these instances have several transhipment nodes.

***Canad – Bipart***

In Table 3.14 the results for the *Canad – Bipart* instances are given.

The BP program is always faster than Cplex 6.6.

For the instances with more than 10% of saturated arcs (*p2*, *p11*, and *p15*) the BP is considerably faster. The same is true for the two fractional instances (*p11* and *p27*) even when the difference in the number of optimised nodes is very significant (*p11*).

***Canad – Mulgen I***

In Table 3.15 the results for the *Canad – Mulgen I* instances are given.

None of the instances of this set is fractional. The BP program is consistently about three times faster than Cplex 6.6. This is due to the smaller number of constraints of the master problem of the BP approach, since the subproblems are shortest path problems.

***Canad – Mulgen II***

In Table 3.16 the results for the *Canad – Mulgen II* instances are given.

For almost all instances, the BP program is slightly faster than Cplex 6.6. For these instances there are no transhipment nodes. We note the reduced proportion of saturated arcs in an optimal solution. In that case, in general, it is easier to obtain a good first set of columns and the column generation procedure converges rapidly.

***Mnetgen 1***

The results for the *Mnetgen 1* instances are given in Table 3.17. The results presented in each row are relative to the average value of the three instances generated with the same input parameters.

The results of the Cplex 6.6 program are clearly better than those of BP for all instances. We note that the instances of this group do not have any transhipment nodes. The high

proportion of saturated arcs suggests that the BP algorithm starts from a solution far from the optimal, which is confirmed when observing that several initial solutions are not feasible (in particular for the more difficult instances, that is, the groups *4*, *5*, *6*, and *10*, *11*, *12*). We can also note that for the 10-th, 11-th and 12-th instances with 8, 16 and 32 commodities the average number of nodes is very high when compared with Cplex 6.6.

***Mnetgen 2***

The results for the *Mnetgen 2* instances are given in Table 3.18. The results presented in each row are relative to the average value of the three instances generated with the same input parameters.

In this set of instances the BP results are better for the easy instances (1, 2, 3, and 7, 8, 9) and worse for the difficult instances (4, 5, 6, and 10, 11, 12).

***Mnetgen 3***

The results for the *Mnetgen 3* instances are given in Table 3.19. Again the results presented in each row are relative to the average value of the three instances generated with the same input parameters.

The results of the BP program are clearly better than the results of Cplex 6.6 for all instances.

***PDS***

In Table 3.20 the results for the *PDS* instances are given.

These results are perplexing: for some instances BP is much better and for some others it is much worse. To analyse this in depth, it would be necessary to gain some insight into the structure of these instances. We note that these instances have a small number of commodities and a large number of transhipment nodes (in fact the subproblems are shortest path problems).

***Planar***

In Table 3.21 the results for the *Planar* instances are given.

For these instances the BP program was clearly more efficient. Due to memory limits, it was not possible to solve the *planar150* instance with Cplex 6.6. We note that the arc formulation for that instance has 335 850 rows and 1 903 150 columns, while the path formulation has only 3 089 rows and the RMP that gave the fractional optimal solution had 14 944 columns. However, we could not find a feasible integer solution, after optimising 600 nodes of the search tree.

**Conclusions of the comparative computational tests**

The linear relaxation of almost all instances chosen to test our branch-and-price algorithm had an integral optimal solution, which does not allow drawing definitive conclusions about the efficiency of the proposed approach. However, noting that in some large instances the arc formulation requires prohibitive amounts of memory, we can conclude that, at least for those instances, our approach is clearly adequate for the integer MFP.

For the instances *Aertranspo*, *Canad*, *Mnetgen 3*, and *Planar*, the branch-and-price algorithm performed better than Cplex 6.6. That is due to the structure of some of those instances (which include several transhipment nodes and/or shortest path subproblems) and to the quality of the initial solution, which was the case for instances with a small percent of saturated arcs in an optimal solution. For the *Mnetgen 1* instances the BP program was clearly inefficient: the tightness of the capacity constraints and the inexistence of transhipment nodes made these instances very difficult for the BP program to solve. For the *Mnetgen 2* and *PDS* instances the results were balanced.

| *Instance* | *Saturated arcs (%)* | *Gap* | *Time* | | *Nodes* | |
|---|---|---|---|---|---|---|
| | | | *BP* | *CPLEX* | *BP* | *CPLEX* |
| *jl23* | 22.5 | 0.00 | **0.1** | 0.2 | 1 | 1 |
| *jl049* | 23.4 | 0.00 | **0.3** | 0.7 | 1 | 1 |
| *jl141* | 15.8 | 0.00 | **7.0** | 28.5 | 1 | 1 |
| *jl147* | 21.5 | 0.00 | **17.6** | 91.2 | 1 | 1 |
| *jl158* | 13.6 | 0.00 | **8.0** | 32.2 | 1 | 1 |
| *jl188* | 3.3 | 0.00 | **5.7** | 35.8 | 1 | 1 |
| *jl207* | 29.3 | 0.00 | **169.7** | 265.1 | 1 | 1 |
| *jl209* | 30.1 | 0.00 | **126.8** | 289.6 | 2 | 1 |

Table 3.13 Results for the *Aertranspo* instances.

| *Instance* | *Saturated arcs (%)* | *Gap* | *Time* | | *Nodes* | |
|---|---|---|---|---|---|---|
| | | | *BP* | *CPLEX* | *BP* | *CPLEX* |
| *p01* | 4.0 | 0.00 | **0.2** | 0.3 | 1 | 1 |
| *p03* | 20.5 | 0.00 | **0.4** | 2.7 | 1 | 3 |
| *p05* | 0.5 | 0.00 | **2.4** | 2.5 | 1 | 1 |
| *p07* | 5.0 | 0.00 | **2.6** | 3.6 | 1 | 1 |
| *p09* | 0.6 | 0.00 | **0.3** | 0.4 | 1 | 1 |
| *p11* | 15.2 | 1.20 | **4.3** | 9.9 | 27 | 4 |
| *p13* | 1.8 | 0.00 | **3.6** | 3.7 | 1 | 1 |
| *p15* | 10.4 | 0.00 | **8.8** | 43.7 | 6 | 1 |
| *p17* | 1.8 | 0.00 | **1.0** | 4.1 | 1 | 1 |
| *p19* | 6.6 | 0.00 | **1.2** | 1.8 | 1 | 1 |
| *p21* | 0.1 | 0.00 | **9.1** | 10.9 | 1 | 1 |
| *p23* | 0.2 | 0.00 | **9.5** | 11.7 | 1 | 1 |
| *p25* | 0.0 | 0.00 | **1.6** | 2.0 | 1 | 1 |
| *p27* | 4.0 | 0.50 | **3.0** | 11.4 | 2 | 3 |
| *p29* | 0.2 | 0.00 | **13.9** | 17.7 | 1 | 1 |
| *p31* | 0.6 | 0.00 | **14.9** | 17.1 | 1 | 1 |

Table 3.14 Results for the *Canad − Bipart* instances.

| *Instance* | *Saturated arcs (%)* | *Gap* | *Time* | | *Nodes* | |
|---|---|---|---|---|---|---|
| | | | *BP* | *CPLEX* | *BP* | *CPLEX* |
| *p33* | 0.4 | 0.00 | **0.2** | 0.9 | 1 | 1 |
| *p35* | 3.5 | 0.00 | **0.2** | 0.4 | 1 | 1 |
| *p37* | 0.4 | 0.00 | **0.9** | 2.5 | 1 | 1 |
| *p39* | 2.6 | 0.00 | **0.9** | 3.1 | 1 | 1 |
| *p41* | 0.3 | 0.00 | **0.2** | 0.6 | 1 | 1 |
| *p43* | 3.5 | 0.00 | **0.2** | 0.6 | 1 | 1 |
| *p45* | 2.8 | 0.00 | **1.1** | 3.1 | 1 | 1 |
| *p47* | 3.8 | 0.00 | **1.1** | 3.2 | 1 | 1 |
| *p49* | 1.0 | 0.00 | **1.0** | 2.8 | 1 | 1 |
| *p51* | 3.7 | 0.00 | **0.9** | 2.8 | 1 | 1 |
| *p53* | 0.8 | 0.00 | **4.0** | 11.5 | 1 | 1 |
| *p55* | 2.3 | 0.00 | **4.0** | 12.3 | 1 | 1 |
| *p57* | 0.1 | 0.00 | **1.0** | 3.5 | 1 | 1 |
| *p59* | 0.9 | 0.00 | **1.1** | 3.5 | 1 | 1 |
| *p61* | 0.3 | 0.00 | **4.7** | 17.2 | 1 | 1 |
| *p63* | 1.5 | 0.00 | **4.7** | 15.2 | 1 | 1 |

Table 3.15 Results for the *Canad − Mulgen I* instances.

| *Instance* | *Saturated arcs (%)* | *Gap* | *Time* | | *Nodes* | |
|---|---|---|---|---|---|---|
| | | | *BP* | *CPLEX* | *BP* | *CPLEX* |
| *p65* | 2.2 | 0.00 | **0.3** | 0.4 | 1 | 1 |
| *p67* | 3.0 | 0.50 | **0.7** | 1.2 | 2 | 1 |
| *p69* | 3.1 | 0.00 | 2.9 | **2.8** | 1 | 1 |
| *p71* | 5.2 | 0.00 | **3.0** | 3.1 | 1 | 1 |
| *p73* | 0.7 | 0.00 | **0.4** | 0.6 | 1 | 1 |
| *p75* | 1.7 | 0.00 | 0.6 | 0.6 | 1 | 1 |
| *p77* | 0.0 | 0.00 | **2.3** | 3.0 | 1 | 1 |
| *p79* | 3.8 | 0.00 | **2.9** | 3.3 | 1 | 1 |
| *p81* | 2.5 | 0.00 | 3.2 | 3.0 | 1 | 1 |
| *p83* | 4.1 | 0.00 | **2.8** | 3.5 | 1 | 1 |
| *p85* | 1.0 | 0.00 | **11.9** | 12.0 | 1 | 1 |
| *p87* | 4.8 | 0.00 | **15.2** | 18.0 | 1 | 1 |
| *p89* | 0.9 | 0.00 | **2.9** | 3.8 | 1 | 1 |
| *p91* | 4.3 | 0.00 | **3.3** | 4.1 | 1 | 1 |
| *p93* | 1.5 | 0.00 | **12.3** | 20.9 | 1 | 1 |
| *p95* | 4.9 | 0.00 | **16.0** | 28.3 | 1 | 1 |

Table 3.16 Results for the *Canad − Mulgen II* instances.

| *Instances* | *Saturated arcs (%)* | *Gap* | *Time* | | *Nodes* | |
|---|---|---|---|---|---|---|
| | | | *BP* | *CPLEX* | *BP* | *CPLEX* |
| *4-1,2,3* | 6.7 | 0.00 | 4.3 | **0.4** | 1.0 | 1.0 |
| *4-4,5,6* | 13.9 | 0.08 | 10.3 | **1.8** | 6.3 | 1.7 |
| *4-7,8,9* | 3.9 | 0.00 | 3.5 | **1.0** | 1.0 | 1.0 |
| *4-10,11,12* | 8.5 | 0.07 | 11.5 | **6.6** | 4.0 | 2.3 |
| *8-1,2,3* | 7.0 | 0.00 | 11.2 | **2.5** | 2.7 | 2.0 |
| *8-4,5,6* | 14.2 | 0.08 | 37.5 | **4.2** | 5.7 | 2.3 |
| *8-7,8,9* | 6.4 | 0.00 | 11.9 | **2.0** | 1.0 | 1.0 |
| *8-10,11,12* | 13.9 | 0.47 | 554.2 | **67.7** | 70.7 | 12.7 |
| *16-1,2,3* | 6.8 | 0.00 | 24.3 | **4.8** | 1.0 | 1.7 |
| *16-4,5,6* | 17.1 | 0.38 | 225.6 | **27.6** | 9.0 | 2.7 |
| *16-7,8,9* | 6.4 | 0.33 | 35.1 | **14.0** | 1.7 | 2.0 |
| *16-10,11,12* | 14.6 | 0.49 | 1686.7 | **135.3** | 128.3 | 7.3 |
| *32-1,2,3* | 9.0 | 0.00 | 121.9 | **21.1** | 1.7 | 2.3 |
| *32-4,5,6* | 17.5 | 0.28 | 1189.9 | **48.8** | 4.0 | 4.3 |
| *32-7,8,9* | 4.3 | 0.00 | 70.6 | **10.5** | 1.0 | 1.0 |
| *32-10,11,12* | 12.8 | 0.44 | 2238.5 | **223.8** | 65.0 | 19.0 |

Table 3.17 Average results for the *Mnetgen 1* instances.

| *Instances* | *Saturated arcs (%)* | *Gap* | *Time* | | *Nodes* | |
|---|---|---|---|---|---|---|
| | | | *BP* | *CPLEX* | *BP* | *CPLEX* |
| *4-1,2,3o* | 1.0 | 0.00 | **0.4** | 0.6 | 1.0 | 1.0 |
| *4-4,5,6o* | 2.4 | 0.00 | **0.5** | 0.7 | 1.0 | 1.0 |
| *4-7,8,9o* | 0.9 | 0.00 | **0.5** | 0.9 | 1.0 | 1.0 |
| *4-10,11,12o* | 1.3 | 0.00 | **0.6** | 1.1 | 1.0 | 1.0 |
| *8-1,2,3o* | 1.7 | 0.00 | **0.9** | 2.0 | 1.0 | 1.0 |
| *8-4,5,6o* | 5.1 | 0.00 | 4.5 | 3.4 | 1.0 | 1.0 |
| *8-7,8,9o* | 1.0 | 0.00 | **0.9** | 1.9 | 1.0 | 1.0 |
| *8-10,11,12o* | 2.7 | 0.00 | **2.1** | 2.3 | 1.0 | 1.0 |
| *16-1,2,3o* | 2.3 | 0.00 | **1.9** | 4.4 | 1.0 | 1.0 |
| *16-4,5,6o* | 8.8 | 0.44 | 32.1 | **28.4** | 4.0 | 2.0 |
| *16-7,8,9o* | 1.0 | 0.00 | **1.7** | 4.5 | 1.0 | 1.0 |
| *16-10,11,12o* | 3.0 | 0.00 | 9.0 | **8.2** | 1.3 | 1.0 |
| *32-1,2,3o* | 5.3 | 0.00 | 11.1 | **10.7** | 1.0 | 1.0 |
| *32-4,5,6o* | 7.9 | 0.00 | 41.3 | **12.9** | 1.0 | 1.0 |
| *32-7,8,9o* | 0.7 | 0.00 | **2.6** | 7.9 | 1.0 | 1.0 |
| *32-10,11,12o* | 4.4 | 0.00 | 69.4 | **18.0** | 1.3 | 1.0 |
| *64-1,2,3o* | 2.4 | 0.00 | **13.0** | 15.0 | 1.0 | 1.0 |
| *64-4,5,6o* | 8.4 | 0.00 | 200.9 | **26.3** | 1.0 | 1.0 |
| *64-7,8,9o* | 1.7 | 0.00 | **8.5** | 16.1 | 1.0 | 1.0 |
| *64-10,11,12o* | 6.8 | 0.00 | 51.3 | **38.4** | 1.0 | 1.0 |
| *128-1,2,3o* | 3.4 | 0.00 | **27.3** | 36.7 | 1.0 | 1.0 |
| *128-4,5,6o* | 8.8 | 0.00 | 664.8 | **86.3** | 1.0 | 1.0 |
| *128-7,8,9o* | 4.2 | 0.00 | **18.1** | 48.0 | 1.0 | 1.0 |
| *128-10,11,12o* | 8.9 | 0.00 | 827.2 | **56.6** | 1.0 | 1.0 |
| *256-1,2,3o* | 4.2 | 0.00 | **81.9** | 87.2 | 1.0 | 1.0 |
| *256-4,5,6o* | 8.7 | 0.00 | 940.5 | **166.9** | 1.0 | 1.0 |
| *256-7,8,9o* | 5.0 | 0.00 | **48.1** | 149.1 | 1.0 | 1.0 |
| *256-10,11,12o* | 8.0 | 0.00 | **194.2** | 240.5 | 1.0 | 1.0 |

Table 3.18 Average results for the *Mnetgen 2* instances.

| *Instances* | *Saturated arcs (%)* | *Gap* | *Time* | | *Nodes* | |
|---|---|---|---|---|---|---|
| | | | *BP* | *CPLEX* | *BP* | *CPLEX* |
| *4-1,2,3c* | 0.9 | 0.00 | **0.2** | 0.3 | 1.0 | 1.0 |
| *4-4,5,6c* | 2.4 | 0.00 | 0.3 | 0.3 | 1.0 | 1.0 |
| *4-7,8,9c* | 0.7 | 0.00 | **0.2** | 0.3 | 1.0 | 1.0 |
| *4-10,11,12c* | 2.2 | 0.00 | 0.4 | 0.4 | 1.0 | 1.0 |
| *8-1,2,3c* | 1.5 | 0.00 | **0.3** | 0.6 | 1.0 | 1.0 |
| *8-4,5,6c* | 4.6 | 0.11 | **0.5** | 2.3 | 1.3 | 1.3 |
| *8-7,8,9c* | 1.1 | 0.00 | **0.4** | 0.7 | 1.0 | 1.0 |
| *8-10,11,12c* | 2.7 | 0.00 | **0.6** | 1.0 | 1.0 | 1.0 |
| *16-1,2,3c* | 1.8 | 0.00 | **0.7** | 1.4 | 1.0 | 1.0 |
| *16-4,5,6c* | 3.5 | 0.00 | **0.9** | 1.9 | 1.0 | 1.0 |
| *16-7,8,9c* | 0.9 | 0.00 | **0.6** | 1.4 | 1.0 | 1.0 |
| *16-10,11,12c* | 3.6 | 0.67 | **1.3** | 6.1 | 2.3 | 1.3 |
| *32-1,2,3c* | 5.3 | 0.00 | **1.5** | 4.3 | 1.0 | 1.0 |
| *32-4,5,6c* | 11.5 | 0.17 | **4.2** | 15.7 | 6.7 | 1.3 |
| *32-7,8,9c* | 2.0 | 0.00 | **1.2** | 4.3 | 1.0 | 1.0 |
| *32-10,11,12c* | 4.4 | 3.44 | **2.0** | 16.8 | 2.3 | 2.7 |
| *64-1,2,3c* | 1.9 | 0.00 | **2.0** | 9.2 | 1.0 | 1.0 |
| *64-4,5,6c* | 10.7 | 0.00 | **4.6** | 26.1 | 1.0 | 1.0 |
| *64-7,8,9c* | 2.1 | 0.00 | **1.9** | 10.6 | 1.0 | 1.0 |
| *64-10,11,12c* | 6.5 | 0.00 | **2.8** | 16.9 | 1.0 | 1.0 |
| *128-1,2,3c* | 2.9 | 0.00 | **3.8** | 32.1 | 1.0 | 1.0 |
| *128-4,5,6c* | 15.6 | 0.11 | **18.2** | 155.3 | 7.7 | 1.3 |
| *128-7,8,9c* | 3.0 | 0.00 | **4.1** | 30.4 | 1.0 | 1.0 |
| *128-10,11,12c* | 4.6 | 0.00 | **4.0** | 50.7 | 1.0 | 1.0 |
| *256-1,2,3c* | 5.0 | 0.00 | **7.5** | 79.6 | 1.0 | 1.0 |
| *256-4,5,6c* | 7.8 | 0.00 | **9.6** | 111.2 | 1.0 | 1.0 |
| *256-7,8,9c* | 2.4 | 0.00 | **6.5** | 102.2 | 1.0 | 1.0 |
| *256-10,11,12c* | 13.2 | 0.00 | **13.3** | 201.6 | 1.0 | 1.0 |

Table 3.19 Average results for the *Mnetgen 3* instances.

| *Instance* | *Saturated arcs (%)* | *Gap* | *Time* | | *Nodes* | |
|---|---|---|---|---|---|---|
| | | | *BP* | *CPLEX* | *BP* | *CPLEX* |
| *pds1* | 9.1 | 0.00 | **0.2** | 0.3 | 1 | 1 |
| *pds2* | 7.2 | 0.00 | **0.3** | 0.8 | 1 | 1 |
| *pds3* | 6.2 | 0.00 | **0.8** | 3.2 | 1 | 1 |
| *pds4* | 5.3 | 0.00 | **1.3** | 2.9 | 1 | 1 |
| *pds5* | 4.8 | 0.00 | **2.0** | 3.8 | 1 | 1 |
| *pds6* | 4.6 | 0.00 | **2.9** | 5.2 | 1 | 1 |
| *pds7* | 4.7 | 0.00 | **4.4** | 5.9 | 1 | 1 |
| *pds8* | 4.6 | 0.00 | **4.8** | 6.8 | 1 | 1 |
| *pds9* | 4.5 | 0.00 | **6.1** | 8.7 | 1 | 1 |
| *pds10* | 4.3 | 0.00 | **8.0** | 9.3 | 1 | 1 |
| *pds11* | 4.3 | 0.00 | **10.4** | 10.7 | 1 | 1 |
| *pds12* | 4.3 | 0.00 | **11.3** | 13.4 | 1 | 1 |
| *pds13* | 4.3 | 0.00 | 20.8 | **7.5** | 1 | 1 |
| *pds14* | 4.2 | 0.00 | 22.9 | **10.9** | 1 | 1 |
| *pds15* | 4.1 | 0.00 | 24.9 | **11.3** | 1 | 1 |
| *pds18* | 4.0 | 0.00 | 37.7 | **18.3** | 1 | 1 |
| *pds20* | 3.9 | 0.00 | 54.9 | **30.6** | 1 | 1 |
| *pds21* | 3.8 | 0.00 | 62.9 | **24.8** | 1 | 1 |
| *pds24* | 3.8 | 0.00 | 80.3 | **53.8** | 1 | 1 |
| *pds27* | 3.7 | 0.00 | **108.7** | 206.9 | 1 | 1 |
| *pds30* | 3.7 | 0.00 | **151.3** | 852.9 | 1 | 1 |
| *pds33* | 3.7 | 0.00 | **183.3** | 248.3 | 1 | 1 |
| *pds36* | 3.6 | 0.00 | **252.3** | 816.4 | 1 | 1 |
| *pds40* | 3.6 | 0.00 | **513.6** | 899.1 | 1 | 1 |
| *pds80* | 3.0 | 0.00 | 1017.1 | **399.4** | 4 | 1 |

Table 3.20 Results for the *PDS* instances.

| *Instance* | *Saturated arcs (%)* | *Gap* | *Time* | | *Nodes* | |
|---|---|---|---|---|---|---|
| | | | *BP* | *CPLEX* | *BP* | *CPLEX* |
| *planar30* | 10.7 | 0.00 | **0.3** | 1.1 | 1 | 1 |
| *planar50* | 12.4 | 0.00 | **2.1** | 21.6 | 1 | 1 |
| *planar80* | 24.3 | 0.00 | **23.7** | 323.8 | 4 | 1 |
| *planar100* | 16.4 | 0.00 | **48.5** | 1501.9 | 2 | 1 |
| *planar150* | 27.8 | * | 1484.2* | ** | * | ** |

Table 3.21 Results for the *Planar* instances.
* An integer solution was not found after optimising 600 nodes of the BP tree.
** A fractional optimal solution was not obtained due to excessive memory requirements.

## 3.5 Conclusions

In this Chapter we presented a branch-and-price approach to the minimum cost integer multicommodity flow problem. The developed algorithm is based on a path formulation derived for general instances (where commodities can have multiple origins and destinations) of the problem.

The proposed approach is based on branching on the arc variables, taking into account that, in the nodes of the branch-and-price tree, other than the root, the subproblems (shortest path problems) may have negative cost cycles. Our algorithm deals efficiently with that possibility by explicitly introducing those cycles in the (restricted) master problem. This approach can be also used in other multicommodity flow problems (such as non-oriented problems, multicommodity maximal flow problems or multicommodity flow problems with extra constraints on paths – of which hop constraints are an example).

Computational tests allowed us to compare different versions of the algorithm and to compare their results with a general purpose solver (Cplex 6.6) optimising the arc formulation. Conclusions of the computational tests were not so expressive as we expected, since the linear relaxation of almost all the instances tested had an integral optimal solution. Anyhow, the proposed algorithm provided better time results in several instances, and, for the larger ones, it can be concluded that it is the only feasible approach to be followed, given the huge memory requirements of the formulation that are needed by a general-purpose solver.

Improvements can still be made, such as a more judicious choice of the branching variable and a more sophisticated search strategy. Also, more effective branching rules can be devised. With our approach, as long as branching rules are derived in the arc variables, they are compatible with the subproblem in all nodes of the branch-and-price tree. This allows their extension to branching rules based on several (arc) variables.

# 4 Binary Multicommodity Flow Problem

In this Chapter, we address branch-and-price algorithms for the binary multicommodity flow problem. This problem is defined over a capacitated network in which we intend to route a set of commodities, each one with a given origin, destination and demand, at minimal cost, without exceeding the arc capacities. Furthermore, the flow of each commodity must be routed using a single path.

Formulating this problem with decision variables representing flows on each arc for each commodity gives rise to a large linear (binary) program with two types of constraints: flow conservation constraints and capacity constraints. Based on the Dantzig-Wolfe decomposition principle we obtain two different decompositions (path and knapsack) depending on which type of constraints define the subproblem. In order to solve the binary problem, we combine column generation and branch-and-bound, developing branching rules that preserve the structure of the subproblem in the branch-and-bound tree for both decompositions.

The linear relaxation of the path decomposition provides the same lower bound as the original formulation, and its potential advantage lies in capturing the "network with additional constraints" structure of the problem. The knapsack decomposition, in general, provides better lower bounds but does not explore the aforementioned structure of the problem.

For the path decomposition, we compare the developed branching rule with one previously presented in (Barnhart et al., 2000). We present computational results for the two decompositions, and compare them with the ones given by a general-purpose integer programming solver.

## 4.1 Introduction

The binary minimum cost multicommodity flow problem (MFP) is defined over a directed network in which several commodities share the capacity of the arcs in order to be shipped from the origin to the destination nodes. There is a unit cost flow associated with each arc of the network and with each commodity. The minimum cost binary MFP amounts to finding the minimum cost routing of all the commodities, taking into account that the flow of each commodity cannot be split.

Other designations for this problem are common, such as non-bifurcated routing problem, traffic placement problem, single path routing problem, path selection problem or multiple source unsplittable MFP. Most of those designations refer to communication problems. In those applications, the routing of a set of traffic demands between different users is to be decided, taking into account the capacity of the network arcs and the fact that the traffic between each pair of users cannot be split. In (Parker and Ryan, 1994) an example of routing video data is described, and in (Ouaja and Richards, 2004) the binary MFP is described in the context of traffic engineering.

Other applications, such as production planning and distribution/transportation (an example of express package delivery is given in (Barnhart et al., 2000)), may also be considered, whenever a multicommodity flow model is used and the commodities cannot be split.

Most of the work on MFPs is about its linear version, where the demand of a commodity can be split along different paths. References to surveys, applications and solution methods to the linear MFP were given in Chapter 4, where the integer MFP was considered. In that problem, the demand of each commodity can be split but not each unit.

Different approaches for the binary MFP have been proposed: approximation algorithms (for example, (Kolliopoulos and Stein, 1999)), heuristics ((Wang and Wang, 1999; Costa et al., 2002)) and exact methods.

In our present work, we develop two exact methods based on Dantzig-Wolfe decomposition (DWD) (Dantzig and Wolfe, 1960). The binary MFP is formulated by defining the decision variables as the flows in the arcs and with two types of constraints: flow conservation and capacity. In a path decomposition, the subproblem is defined by the flow conservation constraints; in a knapsack decomposition, the subproblem is defined by the capacity constraints. In both cases, we combine column generation and branch-and-bound (branch-and-price) to obtain optimal solutions to the binary MFP. Branch-and-price methods

were reviewed in Chapter 2; other surveys can be found in (Barnhart et al., 1998; Wilhelm, 2001; Lübbecke and Desrosiers, 2002).

For the path decomposition, the approach presented here follows the one introduced in (Barnhart et al., 2000), where cuts are also incorporated in the solution procedure (branch-and-price-and-cut). We make a slight extension by using general lifted cover inequalities (instead of simple lifted cover inequalities), but our main contribution is the development of a different branching scheme, which may be seen as the fundamental issue when combining column generation and branch-and-bound. The main potential advantage of the path decomposition is that the size of the linear programs solved in a column generation scheme is, in general, considerably smaller than the size of the original linear program. In addition, the network structure of the problem is explored: the binary MFP can be seen as a set of shortest path problems (defined by the 'easy constraints', using a usual decomposition terminology), one for each commodity, and additional constraints that relate them (the 'hard constraints').

A branch-and-price algorithm based on the knapsack decomposition is here developed and tested, to our best knowledge, for the first time. Similar approaches were used before to obtain lower bounds to network design problems (Holmberg and Yuan, 2000; Crainic et al., 2001), but with different solution approaches from the one introduced here. The main potential advantage of this decomposition is that its lower bound is, in general, tighter than the given by the linear relaxation of the original formulation or by the path decomposition.

Other exact methods have been described in the literature, some of them developed at the same time of ours (Alvelos and Carvalho, 2003). In (Belaidouni and Ben-Ameur, 2003) super additive cuts are used to strengthen the formulation based on flows in arcs; in (Park et al., 2003) a column generation formulation with two types of columns is combined with branch-and-bound; in (Ouaja and Richards, 2003; Ouaja and Richards, 2004) a subgradient algorithm (for the Lagrangean relaxation of the capacity constraints or, equivalently, the path decomposition) is combined with constraint logic programming. In (Parker and Ryan, 1994) and (Park et al., 1996), a related problem is treated. We will discuss with more detail some of those approaches when describing our work. In none of those references, the methods developed were compared with a state-of-the-art general-purpose solver, as they are in the present work.

This Chapter is organised as follows. In the next Section, we formally describe the binary MFP and a formulation based on flows in arcs. That formulation is the base for the decompositions that will be introduced in the following two Sections. Section 4.3 is devoted to the path decomposition with a particular emphasis on branching rules and the use of lifted cover inequalities. In Section 4.4, the knapsack decomposition is introduced, the branch-and-price algorithm is described, and some variants are discussed in order to improve its computational

efficiency. In Section 4.5, computational results of both decompositions and of a general-purpose solver (Cplex 8.1 (ILOG, 2002)) are given. Finally, in Section 4.6, we present the main conclusions of this work.

## 4.2 Problem Definition and Original Formulation

We consider a network formed by a set of $n$ nodes, represented by $N$, and a set of $m$ arcs, represented by $A$. We use an index $i=\{1,...,n\}$ to represent a node and a pair of indices $ij$ to represent an arc which has origin in node $i$ and destination in node $j$. We define a set $K$ of $h$ commodities, indexed by $k$. Each commodity $k$ is characterised by an origin, $o^k$, a destination, $d^k$, and an integer demand, $r^k$, which is the number of units that are supplied at its origin and that are required at its destination. We also define an integer capacity $u_{ij}$ associated with each arc of the network and a unit cost, $c_{ij}^k$, associated with the flow of commodity $k$ on arc $ij$. We make the usual assumption, $c_{ij}^k \geq 0, \ \forall ij \in A, \ \forall k \in K$.

The binary MFP consists in finding the minimum cost routing of all the demand of all the commodities taking into account that the demand of each commodity cannot be split.

The original formulation is obtained using decision variables that represent the proportion of the demand of each commodity that flows in each arc. Forcing those variables to be binary is the same as forcing every flow of every commodity to be routed along a single path.

The decision variables are represented as $x_{ij}^k$. The original formulation is as follows.

$$Min \sum_{k \in K} \sum_{ij \in A} c_{ij}^k r^k x_{ij}^k \qquad (OB)$$

*subject to:*

$$\sum_{j:ij \in A} x_{ij}^k - \sum_{j:ji \in A} x_{ji}^k = \begin{cases} 1, if \ i = o^k \\ -1, if \ i = d^k \\ 0, if \ i \neq o^k, i \neq d^k \end{cases}, \forall i \in N, \forall k \in K \qquad (4.1)$$

$$\sum_{k \in K} r^k x_{ij}^k \leq u_{ij}, \ \forall ij \in A \qquad (4.2)$$

$$x_{ij}^k \in \{0,1\}, \ \forall k \in K, \ \forall ij \in A. \qquad (4.3)$$

Constraints *(4.1)* are flow conservation constraints. They state that, for each commodity, the difference between the flow that enters and the flow that leaves each node is equal to the supply/demand of that same node. Constraints *(4.2)* are capacity constraints. They state that the total flow on each arc must be less than or equal to its capacity.

In this work, we apply two decompositions to the formulation *(OB)* and solve them by a column generation based algorithm. We now present a slightly different formulation that, potentially, is more adequate to be solved by column generation.

Column generation algorithms can also be viewed as cutting planes methods in the dual space (as detailed in Chapter 2). Primal degeneracy leads to the slower convergence of the column generation method (as in simplex methods) and corresponds to the presence of multiple *dual* optimal solutions. That can be done by excluding redundant primal constraints and turning primal equality constraints into inequalities, of course, as long the obtained model produces an optimal solution that may be used to retrieve an optimal solution to the original problem (Carvalho, 2000).

The original formulation *(OB)* can be modified by removing the flow conservation constraint of each destination node (of all commodities) and by setting the sense of the other flow conservation constraints to "≥". In this way, the following modified original model is obtained.

$$Min \sum_{k\in K}\sum_{ij\in A} c_{ij}^{k} r^{k} x_{ij}^{k} \qquad (MOB)$$

*subject to:*

$$\sum_{j:ij\in A} x_{ij}^{k} - \sum_{j:ji\in A} x_{ji}^{k} \geq \begin{cases} 1, if\ i = o^{k} \\ 0, if\ i \neq o^{k} \end{cases}, \forall i \in N, \forall k \in K$$

$$\sum_{k\in K} r^{k} x_{ij}^{k} \leq u_{ij},\ \forall ij \in A$$

$$x_{ij}^{k} \in \{0,1\},\ \forall k \in K,\ \forall ij \in A.$$

Applying the path and knapsack decompositions to the *(OB)* formulation does not involve substantial differences from their application to the *(MOB)* formulation.

## 4.3 Branch-and-Price-and-Cut for the Path Decomposition

### 4.3.1 Dantzig-Wolfe decomposition

We apply the DWD principle to the original formulation *(OB)* presented in the previous Section, defining the subproblem with constraints *(4.1)* and *(4.3)*. In this way, the subproblem is a set of $h$ shortest path problems, one for each commodity, and thus its feasible region has no extreme rays, and each extreme point of the feasible region of subproblem $k$ is associated with a path from $o^k$ to $d^k$. We denote the set of indices $p$ of all the extreme points of the subproblem of commodity $k$ by $P^k$. We represent a given extreme point, with index $p$ and associated with the subproblem of a commodity $k$, by $y^{pk}$. The entry in that vector corresponding to the original variable $x_{ij}^{k}$ is denoted by $y_{ij}^{pk}$, which takes the value *1* if arc *ij* belongs to the path $p$ of commodity $k$, and *0* otherwise. The cost of one extreme point, $c^{pk}$, is given by

$$c^{pk} = \sum_{ij \in A} y_{ij}^{pk} c_{ij}^{k} \ .$$

Finally, we define the weight variable associated with each extreme point, $y^{pk}$, as $\lambda^{pk}$. According to the DWD principle and relaxing the binary requirements, we get the following master problem.

$$Min \sum_{k \in K} \sum_{p \in P^k} c^{pk} r^k \lambda^{pk} \qquad (PB)$$

*subject to:*

$$\sum_{p \in P^k} \lambda^{pk} = 1, \ \forall k \in K \qquad (4.4)$$

$$\sum_{k \in K} \sum_{p \in P^k} y_{ij}^{pk} r^k \lambda^{pk} \leq u_{ij}, \ \forall ij \in A \qquad (4.5)$$

$$\lambda^{pk} \geq 0, \ \forall k \in K, \ \forall p \in P^k.$$

A decision variable of *(PB)*, $\lambda^{pk}$, can be seen as the proportion of the demand of the commodity $k$ that is routed in path $p$. Constraints *(4.4)* are convexity constraints, forcing the demand of each commodity to be routed. Constraints *(4.5)* are the capacity constraints after the redefinition of variables subjacent to the DWD principle (when there are no extreme rays): a solution to *(PB)* can be represented as a convex combination of the extreme points of the subproblem. Given a solution of *(PB)*, we can recover a solution of *(OB)* by using

$$x_{ij}^{k} = \sum_{p \in P^k} y_{ij}^{pk} \lambda^{pk}, \ \forall ij \in A, \ \forall k \in K. \qquad (4.6)$$

The linear relaxation of the path decomposition just presented gives a lower bound that is equal to the one given by the original formulation, since the subproblem has the integrality property (Geoffrion, 1974) (that is, all extreme points are integer). However, the efficiency of the solution methods of the linear relaxation can be very different. The dimension of the basis in the original formulation is $n.h+m$, while, in the path decomposition, it is $h+m$. Since the basis dimension is a major factor to simplex methods efficiency, we can expect the path decomposition to be more efficient in larger instances, given that we use a column generation scheme to deal with the exponential number of columns.

The integrality of the decision variables can be forced in different ways, as we will discuss in subsection 4.3.3.

We note that this formulation could be obtained directly by defining the variables $\lambda^{pk}$. However, turning explicit that this formulation can be obtained by a DWD provides additional insight.

### 4.3.2 Overview of the branch-and-price-and-cut algorithm

As usual when dealing with path formulations of MFPs, in the resolution of the problem *(PB)* we use column generation and, in order to obtain binary optimal solutions, we combine column generation, branch-and-bound and cuts.

In this subsection, we give an overview of the branch-and-price-and-cut algorithm, first introduced in (Barnhart et al., 2000). The differences of our approach are treated in the next subsections.

The flowchart of the solution method for a node of the branch-and-price-and-cut tree is given in Figure 4.1.

Setting the RMP amounts to considering the branching constraints that define the node. In the root node, the RMP must be initialised by including artificial variables (to avoid infeasibility). In the other nodes of the tree, the RMP can include cuts generated while solving other nodes (since global cuts are used as will be detailed in subsection 4.3.4).

In the root node, the subproblem of a commodity $k$ is

$$z^k = Min \sum_{ij \in A} (c_{ij}^{\,k} + w_{ij})\, r^k x_{ij}^{\,k} - \pi^k,$$

*subject to:*

$$\sum_{j:ij \in A} x_{ij}^k - \sum_{j:ji \in A} x_{ji}^k = \begin{cases} 1, if\ i = o^k \\ -1, if\ i = d^k \\ 0, if\ i \neq o^k, i \neq d^k \end{cases}, \forall i \in N, \forall k \in K$$

$$x_{ij}^{\,k} \in \{0,1\},\ \forall k \in K,\ \forall ij \in A.$$

where $w_{ij}$ is the (nonnegative) dual variable associated with the capacity constraint *(4.5)* of arc $ij$ and $\pi^k$ is the (unrestricted in sign) dual variable associated with the convexity constraint of commodity $k$. Thus, the subproblem of a commodity $k$ consists in determining the shortest path between its origin and its destination in a network with modified costs. If the optimal solution has a negative value, the path is attractive, and its associated column inserted in the RMP.

We note that $\sum_{k \in K} z^k$ is a lower bound to the optimal value of the root node (see 2.2.4, page 23), and thus it is possible to compute a gap in each iteration of the column generation procedure easily. This gap can be useful for three purposes. Firstly, as a stopping criterion in order to obtain optimal solutions within the desired accuracy. Secondly, columns of the RMP with a reduced cost greater than the gap can be removed from it, with the guarantee that they will never be generated again. Thirdly, if the lower bound of a given iteration of the column generation algorithm in any node of tree is greater than or equal to the incumbent value, the node can be pruned. We note that in the computation of the lower bound in nodes other than the root, branching and cutting constraints must be taken into account. The computation of the

lower bound is not straightforward in the branching schemes we will review in the next section, but it is straightforward with the proposed branching rule that we present at the end of that Section.

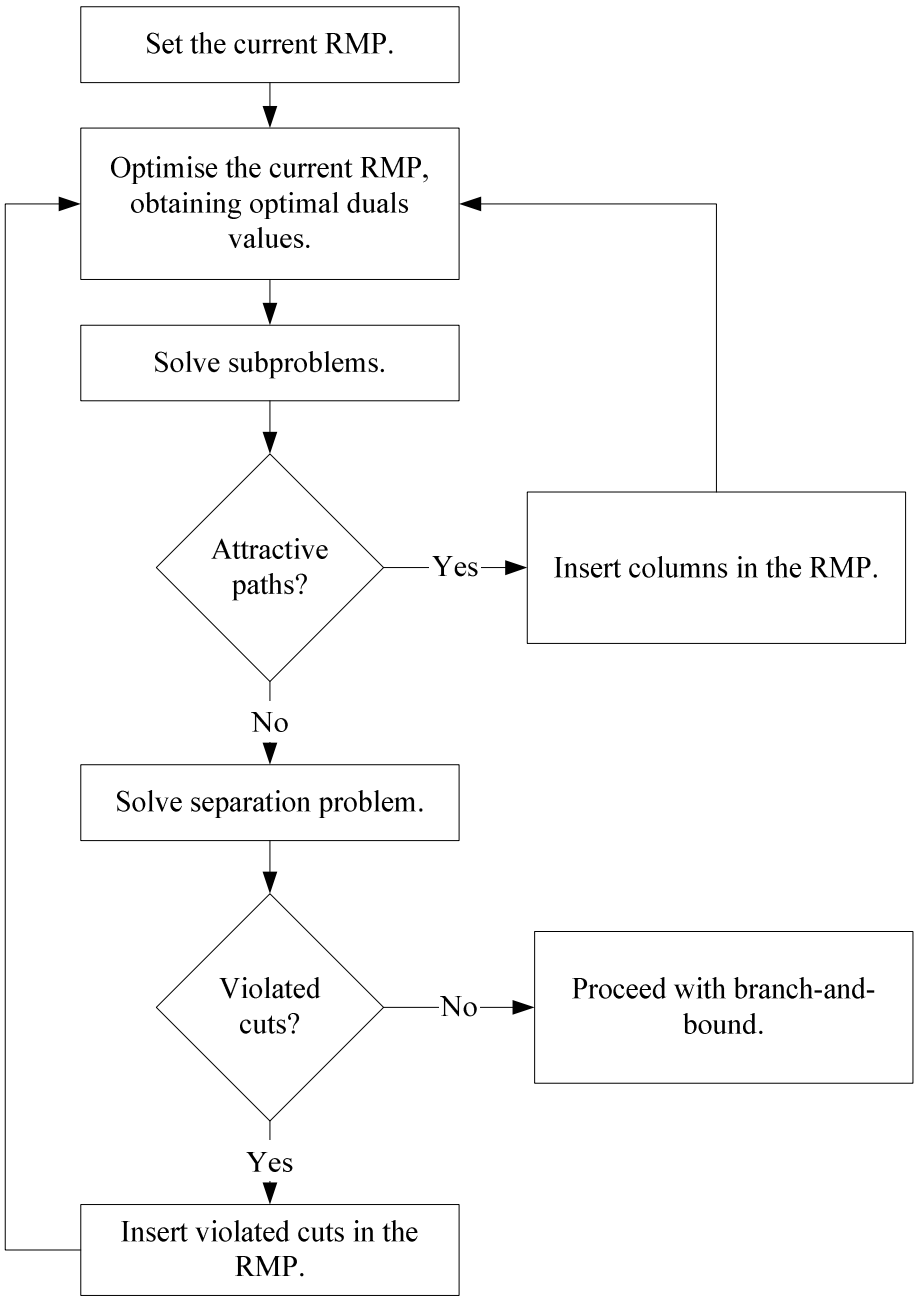



Figure 4.1 Flowchart of the solution method of a node of the search tree.

### 4.3.3 Branching rules

The fundamental issue in a branch-and-price algorithm is the definition of branching rules that allow the exploration of the solution space without compromising the efficiency of the column generation approach to solve the nodes of the tree.

We first review branching rules described in the literature and then propose a new one.

We consider that the flow of a given path $p$ of a given commodity $k$ is fractional, which means that more than one path is being used to route the demand of $k$. Thus, in general, a branch must force path $y^{pk}$ to route all the demand ($\lambda^{pk} = 1$) and the other must force that path $y^{pk}$ not to be used ($\lambda^{pk} = 0$).

In (Parker and Ryan, 1994) and (Park et al., 1996) a different problem is considered, but the branching rules presented there could also be used in the binary MFP. Forcing $\lambda^{pk} = 1$ is trivial. It is sufficient to delete all columns of the RMP that are not associated with path $p$ and not to solve the subproblem of commodity $k$. Forcing $\lambda^{pk} = 0$ implies that path $p$ of commodity $k$ must be excluded from the solution space, which cannot be done in a direct way. Imposing the constraint in the RMP (by deleting its column) does not assure that that path is not (re)generated again by the subproblem.

In (Parker and Ryan, 1994) this issue is overcome by considering several branches. Given $b$ as the number of arcs of path $p$ of commodity $k$, $b+1$ branches are generated. In one branch, $\lambda^{pk}$ is forced to $1$. In each of the others, one of the $l$ arcs of the path is excluded from the shortest path (sub)problem of commodity $k$ (and columns associated with paths with the same characteristic are deleted from the RMP), assuring that, taking them together, the only path excluded is path $p$. Note that it is trivial to exclude a set of arcs from a shortest path problem, but it is not trivial to force the inclusion of a set of arcs. An illustration of this branching rule is given in Figure 4.2.

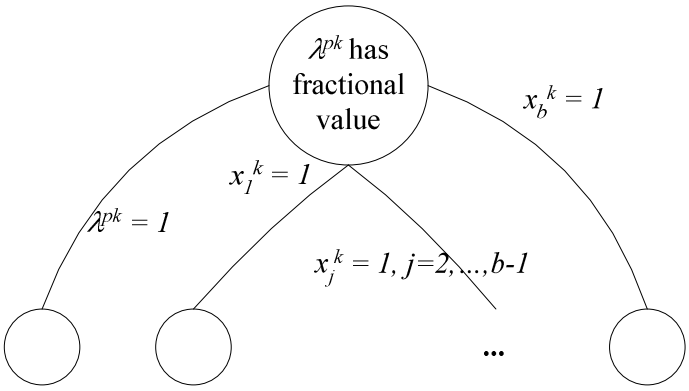



Figure 4.2 Illustration of the (Parker and Ryan, 1994) branching rule.

As noted by the authors of the paper, branching in the arc flow variables has the advantage of taking into account several paths in the same branch, but, on the other hand, the number of nodes to be explored is considerable larger than in a standard branching scheme, where only two branches are created.

In (Park et al., 1996) the regeneration issue is overcome at the expense of turning the subproblem capable of obtaining the second best shortest path (in the case the shortest path is the one to be excluded). The disadvantage of this approach is that in a node of the search tree

where there are *l* paths to be excluded, it may be necessary to determine the *l*-th shortest path. In addition, branching on a single path can lead to unbalanced search trees.

In (Barnhart et al., 2000) another branching rule is proposed. The main idea is based on identifying a node where the flow of a commodity is first split into different paths, and then on branching by forbidding the flow on subsets of arcs that leave that node (avoiding regeneration). This branching rule also avoids the difficulty in solving the shortest path (sub)problem in which some arcs are forced to belong to the solution; the amendment in the subproblem is simply forbidding a set of arcs, which only requires their exclusion from the network.

An illustration of this branching rule is given in Figure 4.3. It is assumed that a commodity has two paths with a flow of *0.5* and all arcs until node *1* are common to those paths (thus the flow in each of those arcs is *1*). In node *1* the flow is split between arcs *12* and *15*. Thus, two branches are created: in one of them, arcs *12* and *13* are excluded, and, in the other, arcs *14* and *15* are excluded. Note that different subsets of arcs could be considered, as long as arcs *12* and *15* were in different subsets.

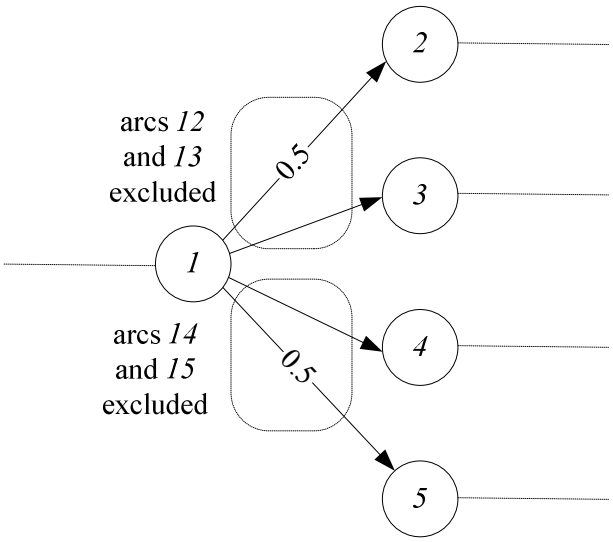



Figure 4.3 Illustration of the (Barnhart et al., 2000) branching rule.

A disadvantage of this branching rule is that a feasible solution may belong to the solution spaces of different nodes of the tree. In the example depicted in Figure 4.3, if the set of paths that carry the flow in the optimal solution does not include any of the arcs that leave node *1*, the branch is irrelevant, since that solution is considered in the solution spaces of both descendant nodes.

The main idea of the branching rule that we propose is to include branching constraints in RMP explicitly and to modify the subproblem accordingly.

Branching constraints are defined in the original variables, given that their values can be easily calculated through *(4.6)* (page 98). Thus, in a branching scheme using only a fractional

variable $x_{ij}^k$, the branches are

$$x_{ij}^k \leq 0 \text{ and } x_{ij}^k \geq 1,$$

or, in the path decomposition variables,

$$\sum_{p \in P^k} y_{ij}^{pk} \lambda_{ij}^{pk} \leq 0 \qquad (4.7)$$

and

$$\sum_{p \in P^k} y_{ij}^{pk} \lambda_{ij}^{pk} \geq 1. \qquad (4.8)$$

The dual variables of these new constraints are then used to modify the costs of the arcs when solving the shortest path subproblems. The modified cost of an arc *ij* for a commodity *k* is now given by

$$r^k w_{ij} + \sum_{u \in U^s} w_{ij}^{ku} - \sum_{l \in L^s} w_{ij}^{kl} + r^k c_{ij}^k,$$

where $w_{ij}^{ku}$ e $w_{ij}^{kl}$ are the dual variables associated with the branching constraints of type *(4.7)* and *(4.8)*, indexed by *u* and *l*, respectively, and $U^s$ and $L^s$ are the set of the indices of the branching constraints that include arc *ij* of commodity *k*.

This branching rule, based on a single fractional original variable, can be extended to branching on several original variables, as exemplified in Figure 4.4.

Any branching rule defined on the original variables can be easily used in this branch-and-price approach.

The disadvantage of this branching rule is that in a node of the branch-and-price tree the modified cost of an arc can be negative, which may imply the existence of a negative cost cycle in the subproblem (of course, if the network has cycles, which may not happen in the instances of some applications).

If a negative cost cycle is identified for some commodity a variable associated with it is inserted in the RMP in the same way as in branch-and-price algorithm for the integer MFP (Chapter 3, subsection 3.3.3, page 70).

In terms of algorithmic modifications, the subproblems in nodes other than the root must be solved by an algorithm that identifies negative cost cycles, such as the labelling correcting ones (for example, (Gallo and Pallottino, 1988)). In the root node, a more efficient labelling setting algorithm can be used.

It is interesting to note that if a cycle variable has a positive value in the optimal solution of a node of the search tree that does not mean that the node can be pruned, something that might be concluded from a first look at this issue, since in an optimal solution all the cycle variables must have zero value. We now give an example that justifies this last observation.

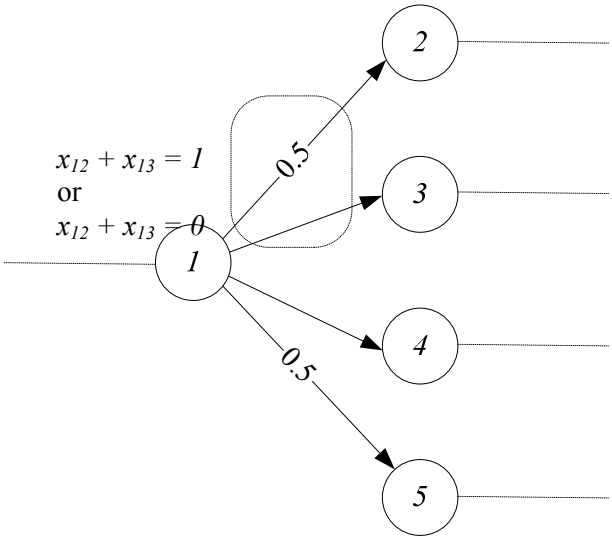



Figure 4.4 Illustration of the proposed branching rule.

**Example 4.1**

Consider Figure 4.5, where a given commodity must be routed from node *1* to node *3*. Consider that in the optimal solution of the problem of a node of the branch-and-bound tree, path *1-2-3* has flow *0.75* and path *1-4-3* has flow *0.25*.

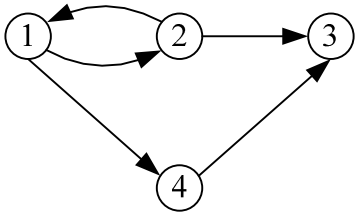



Figure 4.5 Illustration for Example 4.1.

Now consider the branch $x_{12}=1$. One possible optimal solution, for the commodity in question, is the same as before plus a flow of *0.25* on the circuit *1-2-1*. That will happen if the circuit cost is lower than the path *1-4-3* cost.

In terms of flows on arcs, that solution respects the branching constraint. Note also that the branch was effective in the sense that the previous solution (that had $x_{12}=0.75$) was removed from the solution space.

An optimal solution does not have positive flows on cycles. However, the node in question cannot be pruned because the optimal solution for the overall problem can be, for example, routing the commodity through path *1-2-3*. In that case it would be necessary a branch $x_{14}=0$ or $x_{43}=0$ to force the routing exclusively through *1-2-3*.

♦

### 4.3.4 Lifted cover inequalities (LCIs)

Lifted cover inequalities (LCIs) are cuts that may be used to obtain stronger models. Each capacity constraint *(4.2)* of the original model is a knapsack constraint. (Barnhart et al., 2000) incorporated LCIs derived for the knapsack problem in the reformulated model, thus obtaining a branch-and-price-and-cut algorithm.

Here we extend the type of LCIs used in (Barnhart et al., 2000), which are simple LCIs, to the use of general LCIs. A detailed exposition and comparison on different ways of generating LCIs is given in (Gu et al., 1998); here we will concentrate in their application to the binary MFP.

Consider a fractional optimal solution of one node of the search tree, and the capacity constraint of one arbitrary arc *ij* (for simplicity of notation, we omit the arc index)

$$\sum_{k \in C} r^k x^{k*} \leq u,$$

where $x^{k*}$ is calculated through *(4.6)*, and $C$ stands for the indices of the commodities that have a positive flow on arc *ij*.

The set of commodities $C$ is a minimal cover if and only if (iff) $\sum_{k \in C} r^k > u$ (it is a cover) and for each $k' \in C$, $\sum_{k \in C} r^k - r^{k'} \leq u$ (it is minimal). Minimal covers do not necessarily define facets of the associated knapsack polytope. In order to obtain a facet of the knapsack polytope it may be necessary to lift the cover inequality.

Starting from a minimal cover $C$, we define two disjoint sets $C_1$ and $C_2$, where $C_1 \neq 0$. The following inequality is a facet of the knapsack polytope

$$\sum_{k \in C_1} x^k + \sum_{k \in K \setminus C} \alpha^k x^k + \sum_{k \in C_2} \gamma^k x^k \leq |C_1| - 1 + \sum_{k \in C_2} \gamma^k,$$

where $\alpha^k$ is obtained by up lifting and $\gamma^k$ by down lifting (procedures to perform lifting can be found in (Nemhauser and Wolsey, 1999)).

In our implementation we used the heuristic proposed in (Gu et al., 1998) to try to detect a violated LCI for each saturated arc. Thus, for each saturated arc, we first considered three sets: $L = \{k \in K: x^k = 0\}$, $U = \{k \in K: x^k = 1\}$ and $F = K \setminus (L \cup U)$. Then a minimal cover is constructed based on all the $U$ elements and the elements of $F$ that have the highest values. We define $C_2 = U$ and $C_1 = C \setminus C_2$. The initial cover inequality is defined by the $C_1$ elements. The sequence in which the variables are lifted is as follows: firstly, variables that are not in the cover but have a fractional value are up lifted; secondly, variables belonging to $C_2$ are down lifted; thirdly, variables that are not in the cover and have a zero value are up lifted. The lifting sequence starts from the fractional variables (the more important ones in the sense that they are the cause of the

non integrality of the solution), goes to the variables with value equal to one (that are serious candidates for becoming fractional in a subsequent node) and finishes in the variables with value equal to zero (most of them, in general, will have always that value).

The main difference of this way of generating LCIs, when compared to the one used in (Barnhart et al., 2000), is that our set $C_2$ is not necessarily empty. That seems a good feature because, this way, we can construct LCIs in the basis that some variables have, necessarily, the value one.

We note that this procedure does not guarantee that the lifted cover inequality found is violated by the current solution, but finding the most violated LCI is a NP-hard problem (Gu et al., 1999). We also note that, even if we generate all the possible LCIs, the optimal solution of the node could stay fractional, since with LCIs we are just trying to approximate the polytope of the capacity constraints, not considering the other constraints of the problem. In fact, the idea of applying the knapsack decomposition (next subsection) emerged from using an exact description of that polytope (given by the subproblem in the knapsack decomposition), and that decomposition, although giving a better bound, does not exclude the need to perform branching.

We now discuss the implications of the presence of LCIs in the branch-and-price algorithm. After a node is optimised through column generation, we try to detect violated LCIs associated with all the saturated arcs. For each violated LCI found, a new constraint is inserted in the RMP, which is reoptimised using column generation. The procedure is repeated until no more violated LCIs are detected. Then, according to the branch-and-bound method, a decision is taken (for instance, branching if the optimal solution is fractional and the node cannot be pruned).

A crucial issue is how to integrate the cuts added to the RMP in the column generation procedure. Firstly, the cuts are expressed in terms of the original variables (which are flows on arcs). That issue can be easily dealt with. Consider that one LCI associated with arc $ij$ is inserted in the RMP, and that the coefficient of the $x_{ij}^k$ in the LCI is $\beta$. Then the LCI constraint in the RMP must have the coefficient $\beta$ for all paths that are associated with commodity $k$ and include arc $ij$. Secondly, since the RMP has additional constraints, the subproblem must still be capable of correctly pricing the variables.

We represent the dual variable associated with the $l$-th branching constraint of type "less than or equal to" (corresponding to branch on the arc $ij$ and on the commodity $k$) by $w_{ij}^{kl}$. Similarly, we define $w_{ij}^{ku}$ for the $u$-th branching constraint of type "greater than or equal to". In a node $s$ of the branch-and-price tree, we represent the two sets of branching constraints by $L^s$ and $U^s$. We define the dual variable associated with the $g$-th LCI constraint by $w_{ij}^g$. In a node $s$ of the branch-and-price tree, we represent the set of LCI constraints by $G^s$. We also define $\beta_{ij}^{gk}$ as the coefficient in the LCI with index $g$ (associated with arc $ij$) of the commodity $k$. One path $p$ of one commodity $k$ is attractive iff

$$\sum_{ij \in A} r^k y_{ij}^{pk} (w_{ij} + c_{ij}^k) + \sum_{u \in U^s} y_{ij}^{pk} w_{ij}^{ku} - \sum_{l \in L^s} y_{ij}^{pk} w_{ij}^{kl} + \sum_{g \in G^s} y_{ij}^{pk} \beta_{ij}^{gk} \lambda_{ij}^{g} < \pi^k.$$

In this way, the subproblem remains a shortest path for each commodity, and the presence of LCIs constraints does not destroy its structure.

Given their senses, the LCIs constraints have a positive contribution to the modified costs of the arcs, and thus they do not contribute to the existence of negative cost cycles.

Any type of valid inequalities could be used, as long as they were defined in terms of the original variables and inserted in the RMP in the same manner as the LCIs with the same type of modifications in the subproblem.

A last issue about the branch-and-price-and-cut algorithm is that the cuts inserted in one of nodes of the tree are feasible for all nodes of the tree (global cuts). That is achieved in the construction of each of the LCIs. In an arbitrary node of the tree, we start from a minimal cover (that is a valid inequality for all the nodes), then all the variables that have value one are down lifted (assuring the validity for all the nodes of the tree) and the remaining variables are up lifted (strengthening the constraint).

## 4.4 Branch-and-Price for the Knapsack Decomposition

### 4.4.1 Dantzig-Wolfe decomposition

In the knapsack decomposition, we apply the DWD principle to the original formulation, defining the subproblem with constraints *(4.2)* and *(4.3)*. In this way, the subproblem is a set of *m* binary knapsack problems, one for each arc, and thus it has no extreme rays, and each extreme point of subproblem of arc *ij* is associated with a set of commodities that, together, can use arc *ij* without exceeding its capacity.

We denote the set of indices *q* of all the extreme points of the subproblem of arc *ij* by $Q^{ij}$. We represent a given extreme point, with index *q* and associated with the subproblem of an arc *ij*, by $y^{qij}$. The entry in that vector corresponding to the original variable $x_{ij}^k$ is denoted by $y_k^{qij}$. Thus, the cost of one extreme point, $y^{qij}$, is given by

$$c^{qij} = \sum_{k \in K} y_k^{qij} r^k c_{ij}^k.$$

Finally, we define the weight variable associated with each extreme point, $y^{qij}$, as $\lambda^{qij}$. According to the DWD principle and relaxing the binary requirements, we get the following master problem:

$$Min \sum_{ij \in A} \sum_{q \in Q^{ij}} c^{qij} \lambda^{qij} \qquad (KB)$$

*subject to:*

$$\sum_{q \in Q^{ij}} \lambda^{qij} \leq 1, \ \forall ij \in A \qquad (4.9)$$

$$\sum_{j:ij \in A} \sum_{q \in Q^{ij}} y_k^{qij} \lambda^{qij} - \sum_{j:ji \in A} \sum_{q \in Q^{ji}} y_k^{qji} \lambda^{qji} = \begin{cases} 1 \text{ if } i = o^k \\ -1 \text{ if } i = d^k \\ 0 \text{ if } i \neq o^k, i \neq d^k \end{cases}, \forall i \in N, \forall k \in K \qquad (4.10)$$

$\lambda^{qij} \geq 0, \ \forall ij \in A, \ \forall q \in Q^{ij}$.

The decision variables of this (re)formulation, $\lambda^{qij}$, are associated with the selection of the set of commodities that use each arc, being the alternatives the possible combinations of commodities, defined by $y^{qij}$. Constraints *(4.9)* are convexity constraints: at most one set of commodities will use each arc. We note that the origin is an extreme point for each subproblem (associated with not using the arc at all), which justifies the sense of the convexity constraints.

Constraints *(4.10)* are the flow conservation constraints after the redefinition of variables subjacent to the DWD principle.

Given a solution of *(KB)*, we can recover a solution of *(OB)* by using

$$x_{ij}^k = \sum_{q \in Q^{ij}} y_k^{qij} \lambda^{qij}, \ \forall ij \in A, \ \forall k \in K. \qquad (4.11)$$

The number of constraints of this (re)formulation is the same as the number of constraints of the original formulation. The number of decision variables is much larger (one decision variable for each arc and for each possible combination of commodities that use the arc).

Being so, in general, this decomposition approach does not reduce the size of the problems to be solved. However, the lower bounds it gives are, in general, tighter than the ones given by the original formulation or by the path decomposition, since the subproblems are binary knapsack problems that do not have the integrality property. The original formulation or the path decomposition consider that the flow on each arc may be fractional, and thus each capacity constraint can be seen as a continuous knapsack constraint. In this decomposition, all solutions that are feasible to the continuous knapsack constraint, but are not feasible considering binary variables are excluded. These solutions are the same that we exclude when using lifted cover inequalities in the path decomposition. In that case, the LCIs define facets of the polytope defined by the binary knapsack constraints. In this case, those facets are implicitly defined by taking convex combinations of the binary extreme points of the same polytope.

### 4.4.2 Solving the root node

We assume that a set of subproblems' solutions, associated with a set of indices, $\overline{Q}^{ij} \subseteq Q^{ij}, \ \forall ij \in A$, which allows building a feasible solution to the following restricted master problem (RMP), is known.

$$Min \sum_{ij \in A} \sum_{q \in \overline{Q}^{ij}} c^{qij} \lambda^{qij} \qquad (KRMP)$$

*subject to:*

$$\sum_{q \in \overline{Q}^{ij}} \lambda^{qij} \leq 1, \ \forall ij \in A \qquad (4.12)$$

$$\sum_{j:ij \in A} \sum_{q \in \overline{Q}^{ij}} y_k^{qij} \lambda^{qij} - \sum_{j:ji \in A} \sum_{q \in \overline{Q}^{ji}} y_k^{qji} \lambda^{qji} = \begin{cases} 1 \text{ if } i = o^k \\ -1 \text{ if } i = d^k \\ 0 \text{ if } i \neq o^k, i \neq d^k \end{cases}, \forall i \in N, \forall k \in K \qquad (4.13)$$

$\lambda^{qij} \geq 0, \ \forall ij \in A, \ \forall q \in \overline{Q}^{ij}$.

We define $\pi^{ij}$ as the (nonnegative) dual variables of constraints *(4.12)* and $w_i^k$ as the (unrestricted in sign) dual variables of constraints *(4.13)*.

For arc *ij* the subproblem is

$$z^{ij} = Min \sum_{k \in K} (c_{ij}^k r^k - w_i^k + w_j^k) x_{ij}^k - \pi^{ij} \qquad (KSP)$$

*subject to:*

$$\sum_{k \in K} r^k x_{ij}^k \leq u_{ij}$$

$x_{ij}^k \in \{0,1\}, \ \forall k \in K.$

As in the case of the path decomposition (subsection 4.3.2), or in the general application of DWD principle (subsection 2.2.4), the value given by $\sum_{ij \in A} z^{ij}$ is a lower bound to the optimal value of the root node, and thus it is possible to compute a gap in each iteration of the column generation procedure easily. As described for the path decomposition (subsection 4.3.2), this gap can be useful for obtaining optimal solutions within the desired accuracy, for fixing variables (which can be useful when removing columns) or for fathoming nodes as soon as, in a given column generation iteration, the lower bound is larger than the incumbent value in a node of the tree.

We note that in the computation of the lower bound in nodes other than the root, branching must be taken into account, an issue that will be discussed in the next subsection.

As will be shown in the Section 4.5, this decomposition turned out to be difficult to deal with. The columns of the RMP are dense columns (in the worst case, if all commodities use the same arc, one column may have *2h+1* non zeros) and the network structure of the original problem is lost. We used three strategies to try to overcome that issue that we now briefly describe.

Firstly, the first RMP only includes the origin flow conservation constraints. Whenever no columns are generated by the subproblems, violated rows not present in the RMP are

inserted and the same procedure goes on until neither attractive columns nor violated rows are detected. An analogy with branch-and-price-and-cut is direct: the constraints play the role of the cuts and the separation problem consists in passing through all that are not present in the RMP, finding the ones that are currently being violated. The flowchart of Figure 4.1 (page 100), with minor terminology modifications, describes the algorithm.

Secondly, in order to try to keep the size of the RMPs manageable by the LP solver, in some iterations, we perform columns removal in such a way that the number of columns of the RMP is kept approximately equal to $3r$, where $r$ is the total number of rows. The selection of columns to be removed is made by sorting them by decreasing reduced costs, reducing the probability that a removed column will be generated again in a subsequent iteration.

Thirdly, we used an inexact RMP solver. For the column generation algorithm to proceed it only needs the RMP to provide a feasible dual point. In particular, in early iterations, it may not be worth solving the RMP exactly, since the columns generated in an optimal dual point may have poor "quality" anyway (in the sense that they have a small probability of being present in an optimal solution). Thus, the time spent in solving the RMP exactly can be spent in generating more columns that allow reaching a "good" solution (or a feasible one, in a difficult instance) earlier.

### 4.4.3 Branching rules

Branching rules can be defined on the original variables or on the weight variables. We first discuss a branching rule based on the weight variables, noting that if a binary solution is obtained on the weight variables then it corresponds to a binary solution on the original variables. That is not the case in problems with general integer variables.

We branch by creating two nodes: in one of them, one weight variable is set to one and, in the other, the same variable is set to zero.

In the branch where the fractional variable is set to one, all columns associated with different variables of the same arc must be deleted from the RMP. The convexity constraint of the arc must be set to equality. The subproblem of the arc does not need to be solved: there is already one extreme point in the RMP for that arc that has value one.

In the other branch, where the variable is set to zero, we set the cost of the corresponding column to a big value, forcing it to have zero value in the RMP. When solving the subproblem associated with its arc, if the optimal solution is not attractive, or if the column associated with it is attractive but is not yet in the RMP, then there are no changes, when compared with the standard procedure for solving the subproblem. However, if the column is attractive and the corresponding column is already in the RMP, we must find the second best solution. That can be done by using a general procedure for finding the second best solution in a general binary

problem, which consists in adding the following constraint to the problem

$$\sum_{k \in K^1} x^k - \sum_{k \in K \setminus K^1} x^k \leq |K^1| - 1,$$

where $K^1$ is the set of commodities that have value one in the current solution.

This procedure is repeated until we can conclude whether or not the best column not present in RMP is attractive.

One disadvantage of this approach is that the knapsack structure of the subproblem is lost, and thus we cannot use a specialised algorithm to solve it. Branching in the weight variables is not frequently used because, as exemplified here, it leads to major changes in the subproblem. We now turn to branching rules based on the original variables.

The branching rules based on the original variables described for the path decomposition (subsection 4.3.3, page 100) can be used in the knapsack decomposition. The difference is, of course, the calculation of the values of the original variables (through *(4.11)*) and the modifications implied by forcing an arc to have flow of a commodity or not.

If $x_{ij}^k$ has a fractional value, we can derive two branches: $x_{ij}^k = 0$ and $x_{ij}^k = 1$. In both cases, we have to remove some columns related with commodity $k$ and arc $ij$ prior to optimising the resulting node.

In the branch $x_{ij}^k = 0$ we remove the columns associated with arc $ij$ that include commodity $k$ and we also exclude commodity $k$ from the subproblem associated with that arc. This way we guarantee that the optimal solution of the node will have $x_{ij}^k = 0$.

In the branch $x_{ij}^k = 1$ we remove the columns associated with arc $ij$ that do not include commodity $k$ and we force the solution of the subproblem of arc $ij$ to include commodity $k$. That is not enough to guarantee that the total demand of commodity $k$ will flow through arc $ij$. We also have to modify the sense of the convexity constraint of arc $ij$ to an equality. Note that, as opposed to the path decomposition, setting an original variable to one, can be dealt with easily.

The extension to forcing a set of original variables to zero, or to forcing a set of variables to one (which is required by some of the branching rules introduced for the path decomposition) can be easily performed.

The new branching rule introduced in subsection 4.3.3 for the path decomposition can also be used in the knapsack decomposition (in fact, in any branch-and-price algorithm for which an original formulation is known, as detailed in Chapter 2).

By explicitly introducing branching constraints in the master (based on the original variables but, of course, translated into the weight variables) and by taking into account their duals in the objective function of the subproblems, the column generation procedure to solve the problem of each node of the tree suffers minor modifications. The subproblems remain binary knapsack problems. In our implementation of the branch-and-price algorithm, we used this last

branching scheme.

### 4.4.4 Combination of the two decompositions

In (Park et al., 2003) a way of combining the two decomposition approaches presented here is proposed. The main idea is to have a master problem where there are two types of columns: the ones associated with paths and the ones associated with binary knapsack solutions. Being so, there are also two types of subproblems: shortest paths and binary knapsacks. In the RMP, the weight variables associated with each type of subproblems are related by stating that a path of a commodity can only use a given arc if it belongs to the binary knapsack solution of that arc. As described by the authors, the branching rules previously described can easily be applied in that approach, which has the clear advantage of taking the best part of the two decompositions: capturing the network structure (path decomposition) and obtaining good quality lower bounds (knapsack decomposition). The authors describe computational tests that confirm the potential of their approach. A computational comparison of the two decompositions presented here and the approach of (Park et al., 2003) is a natural extension of the computational tests that we made, and will describe in the next Section.

Although in (Park et al., 2003) the contextualisation of the approach in a DWD framework was not treated, the proposed algorithm can be seen as an application of multiple DWD (subsection 2.5.5, page 50).

## 4.5 Computational Results

### 4.5.1 Computational environment and parameters

We performed a set of computational tests for different variants of the branch-and-price algorithms for the path and the knapsack decompositions and compared them with the results obtained by solving the original formulation with a general-purpose solver (Cplex 8.1 (ILOG, 2002)).

The branch-and-price algorithms were implemented using *ADDing – Automatic Dantzig-Wolfe Decomposition for INteger column Generation*, a set of *C++* classes that implements a generic branch-and-price algorithm. In its basic use, the user is only required to provide an original formulation and the decomposition to be used (that is, which constraints define the subproblem(s)), along with some parameters. *ADDing* implements, in a transparent way to the user, a branch-and-price algorithm based on the defined decomposition, returning an optimal solution in the original variables (if there is one, and any stopping criteria specified by the user – for example, maximum number of nodes of the tree – is not achieved). In a more advanced mode, the user may customise parts of *ADDing*, such as the subproblem solver, the use of

subproblem heuristics, and/or the branching rules, as we did.

*ADDing* was implemented using the development environment Microsoft Visual Studio 6.0 and uses Cplex 8.1 for solving the RMPs, and will be presented in Chapter 6.

All the reported results were obtained on a personal computer with a Pentium 4, 2.80 GHz processor, 1 GB of RAM, running Windows XP Professional Edition. All the time values exclude the file input and output and are expressed in seconds. In the case of the Cplex tests, they also exclude the construction of the model.

For the linear relaxations, when solved by Cplex or by *ADDing*, we set the optimality tolerance to *1e−5*. For the integer problems, when solved by Cplex or by *ADDing*, we set the absolute integer gap tolerance to *1e−5*. We used the same tolerance for the integrality of values, that is, a variable is integer if its absolute value differs from an integer by at most *1e−5*. Since, currently, *ADDing* does not have a stopping criterion based on the relative tolerance, we set the Cplex relative tolerance to a very small value (*1e−12*).

### 4.5.2 Test instances

For computational test purposes, we used as test instances the Carbin instances (Alvelos, 2005). These instances were generated by a random instance generator based on the LEDA class library (Mehlhorn and Näher, 1999). The input of the generator is a set of parameters that restrains the characteristics of the instance that will be created, as the number of nodes, arcs, commodities, the minimum and maximum values of the demands, capacities, and costs (which, for each arc, may vary by commodity or not). Some of these values are only indicative values. One feature of the generator is that it always creates a feasible instance (when considering the linear relaxation of the original formulation). Thus, the number of arcs given by the user may be slightly increased, and some capacities may be larger than the maximum value specified. In order to obtain potentially difficult instances, some of the arcs (in particular, those created for feasibility) have a higher probability of getting the maximum cost.

There are 48 *Carbin* instances that can be classified according to five characteristics: the ratio *mean capacity of the arcs / mean demand of the commodities*, the number of commodities, the density of the network, type of costs (for a given arc, being equal for all commodities or not) and variance of costs. All the instances have 32 nodes. They can be divided in two main classes of 24 instances each: for instances with an *s* in their name the ratio *mean capacity / mean demand* is 1.5; for the instances with an *l*, that ratio is 10. In Table 4.1 and in Table 4.2, the characteristics of these 48 instances are presented (replacing *x* by *s* or *l*).

We also tested two *Planar* instances. These instances come from (Larsson and Yuan, 2004) and were obtained in (Frangioni, 2005). Their number of nodes, arcs and commodities are given in Table 4.3.

In the computational tests, we divided the instances in two sets: one of smaller instances (*bs01* to *bs08*, *bl01* to *bl08*, and *planar30*) and another of larger instances.

| *Instance* | *n* | *m* | *h* | *m/n* | *h/n* |
|---|---|---|---|---|---|
| *bx01-bx04* | 32 | 96 | 48 | 3 | 1.5 |
| *bx05-bx08* | 32 | 320 | 48 | 10 | 1.5 |
| *bx09-bx12* | 32 | 96 | 192 | 3 | 6 |
| *bx13-bx16* | 32 | 320 | 192 | 10 | 6 |
| *bx17-bx20* | 32 | 96 | 320 | 3 | 10 |
| *bx21-bx24* | 32 | 320 | 320 | 10 | 10 |

Table 4.1 Characteristics of the *Carbin* instances.

| *Position in the group* | *Maximum cost* | *Arc cost depends on the commodity?* |
|---|---|---|
| *First* | 1000 | No |
| *Second* | 1000 | Yes |
| *Third* | 10 | No |
| *Fourth* | 10 | Yes |

Table 4.2 Characteristics of each group of four consecutive *Carbin* instances

| *Instance* | *planar30* | *planar50* |
|---|---|---|
| *n* | 30 | 50 |
| *m* | 150 | 250 |
| *h* | 92 | 267 |

Table 4.3 Dimensions of the tested *Planar* instances.

### 4.5.3 LCIs and branching rules for the path decomposition

**Implementation issues**

For the path decomposition we implemented the L–2queue algorithm (Gallo and Pallottino, 1988) to solve the shortest path (sub)problems, and used the COIN implementation of the Horowitz-Sahni algorithm (Horowitz and Sahni, 1974) to solve the knapsack problems when constructing LCIs. We used the Cplex 8.1 dual simplex algorithm to solve the RMPs.

When constructing the first RMP we generated columns by two different procedures: solving the subproblems with the original costs, and using a heuristic that sequentially solves the shortest path subproblem of each commodity, deleting the arcs that do not have enough

capacity for the current commodity (of course, updating the capacities according to the paths found previously).

Those procedures do not guarantee that the first RMP has a feasible solution, and thus we add one artificial variable for each convexity constraint.

**LCIs**

The first set of computational tests was meant to compare the use of simple and general LCIs. In Table 4.4 and in Table 4.5, results for the root node of the comparison between no use of LCIs (N columns), simple LCIs (S columns) and general LCIs (G columns), are given for the smaller and larger instances, respectively. The use of simple LCIs translated into an average relative improvement of the lower bound of 3.3%, when compared to not using LCIs.

As expected, on average, using general LCIs is more time-consuming than using simple LCIs. However, for 29 of the 50 instances, it produces a strictly better lower bound (in six instances the lower bound obtained by the use of two types of LCIs is the same). The average relative improvement in the lower bound is 0.02% and the average relative increase of the optimisation time is 25.6%.

In the subsequent computational tests, we used general LCIs in all nodes of the tree.

**Branching rules**

The actual version of *ADDing* has some default branching rules based on the original variables, also allowing the customisation of branching rules according to the problem at hand. Branching is performed by dualising the branching constraints (that is, keeping them in the RMP, taking into account their duals when solving the subproblem). That approach can be used in any integer problem (pure binary, pure integer or mixed integer problems). Although the original implementation of the branching rule of (Barnhart et al., 2000) is made by including the branching constraints in the subproblem (changing the RMP accordingly), branching by dualising the branching constraints leads (theoretically) to the same branch-and-price tree. The only difference is how each node is solved, but the solution space is equal in both cases.

We compared a default branching rule (branching in the variable with the fractional part closest to *0.5*) with three others, for the smaller instances. The number of optimised nodes, for each branching strategy, is given in Table 4.6. The search strategy used (default of *ADDing*) consists in using a depth first search when the node generates descendants and best bound search otherwise.

Column DD refers to the default branching rule with down branching first. Column DU refers to the default branching rule with up branching first. Columns AD (down branching first) and AU (up branching first) refer to the branching variable being the one associated with the arc with more commodities with fractional flow and, among them, the one with greater demand.

Column B refers to the branching rule of (Barnhart et al., 2000). Columns CD (down branching first) and CU (up branching first) refer to the branching rule proposed in subsection 4.3.3 (considering one commodity and a set of arcs leaving the same node).

The first four branching strategies gave a clearly worst result in the last instance (*planar30*).

Comparing CD and CU, CD gave better results in six instances and worse in three instances. Comparing CD and B, CD gave better results in six instances (including the two where more nodes were optimised: *bs07* and *planar30*) and worse in two instances.

| *Instance* | *Time* | | | *Value* | | | *Value Improvement (%)* | |
|---|---|---|---|---|---|---|---|---|
| | *N* | *S* | *G* | *N* | *S* | *G* | *S/N* | *G/S* |
| *bl01* | 0.0 | 0.4 | 0.4 | 1549772.0 | 1608223.6 | 1608868.0 | 3.77 | 0.04 |
| *bl02* | 0.0 | 0.5 | 0.5 | 1641924.0 | 1793115.5 | 1792955.6 | 9.21 | –0.01 |
| *bl03* | 0.1 | 0.6 | 0.5 | 15963.0 | 17246.7 | 17261.7 | 8.04 | 0.09 |
| *bl04* | 0.0 | 1.0 | 1.0 | 17875.0 | 19019.8 | 19059.6 | 6.40 | 0.21 |
| *bl05* | 0.0 | 0.1 | 0.1 | 469918.0 | 474782.0 | 474782.0 | 1.04 | 0.00 |
| *bl06* | 0.0 | 0.1 | 0.1 | 407297.0 | 411480.0 | 411480.0 | 1.03 | 0.00 |
| *bl07* | 0.0 | 0.1 | 0.1 | 5719.0 | 5751.0 | 5751.0 | 0.56 | 0.00 |
| *bl08* | 0.0 | 0.0 | 0.0 | 5658.0 | 5688.0 | 5688.0 | 0.53 | 0.00 |
| *bs01* | 0.0 | 0.5 | 0.5 | 1538239.0 | 1639862.0 | 1639637.4 | 6.61 | –0.01 |
| *bs02* | 0.0 | 0.6 | 0.7 | 1556653.5 | 1694567.8 | 1693413.5 | 8.86 | –0.07 |
| *bs03* | 0.0 | 0.1 | 0.2 | 16593.0 | 16828.0 | 16828.0 | 1.42 | 0.00 |
| *bs04* | 0.1 | 0.6 | 0.6 | 18462.0 | 19605.4 | 19501.9 | 6.19 | –0.53 |
| *bs05* | 0.0 | 0.5 | 0.6 | 459174.0 | 498933.4 | 499341.9 | 8.66 | 0.08 |
| *bs06* | 0.1 | 0.3 | 0.2 | 467310.5 | 500183.9 | 498365.1 | 7.03 | –0.36 |
| *bs07* | 0.0 | 0.6 | 0.5 | 6536.0 | 7145.8 | 7143.7 | 9.33 | –0.03 |
| *bs08* | 0.0 | 0.3 | 0.3 | 6133.0 | 6454.0 | 6427.1 | 5.23 | –0.42 |
| *planar30* | 0.0 | 0.2 | 0.3 | 44350624.0 | 44453752.5 | 44451958.5 | 0.23 | 0.00 |

Table 4.4 Path decomposition computational results: use of LCIs in the root node.
N – No LCIs; S – Simple LCIs; G – General LCIs.

| *Instance* | *Time* | | | *Value* | | | *Value Improvement (%)* | |
|---|---|---|---|---|---|---|---|---|
| | *N* | *S* | *G* | *N* | *S* | *G* | *S/N* | *G/S* |
| *bl09* | 0.1 | 1.9 | 3.0 | 6108239.0 | 6183542.6 | 6192318.2 | 1.23 | 0.14 |
| *bl10* | 0.1 | 1.6 | 1.9 | 6192262.0 | 6243977.6 | 6245926.7 | 0.84 | 0.03 |
| *bl11* | 0.0 | 0.8 | 0.7 | 68302.0 | 68971.0 | 68967.0 | 0.98 | –0.01 |
| *bl12* | 0.1 | 1.1 | 1.4 | 64807.2 | 65647.7 | 65644.8 | 1.30 | 0.00 |
| *bl13* | 0.1 | 6.1 | 7.0 | 3010940.0 | 3118060.2 | 3123924.2 | 3.56 | 0.19 |
| *bl14* | 0.1 | 1.9 | 2.0 | 2324767.0 | 2425133.1 | 2422987.3 | 4.32 | –0.09 |
| *bl15* | 0.1 | 2.6 | 3.0 | 33297.0 | 34182.5 | 34215.9 | 2.66 | 0.10 |
| *bl16* | 0.1 | 1.9 | 2.1 | 26844.0 | 28012.9 | 28013.1 | 4.35 | 0.00 |
| *bl17* | 0.1 | 3.2 | 6.5 | 13086437.0 | 13149043.0 | 13154923.4 | 0.48 | 0.04 |
| *bl18* | 0.1 | 2.9 | 6.2 | 10401389.0 | 10460082.0 | 10467268.8 | 0.56 | 0.07 |
| *bl19* | 0.1 | 3.7 | 8.2 | 108049.0 | 108544.9 | 108602.6 | 0.46 | 0.05 |
| *bl20* | 0.2 | 3.5 | 4.5 | 109612.3 | 110667.9 | 110635.4 | 0.96 | –0.03 |
| *bl21* | 0.2 | 11.2 | 17.3 | 5612118.1 | 5759495.7 | 5770619.3 | 2.63 | 0.19 |
| *bl22* | 0.2 | 7.9 | 9.5 | 4058714.3 | 4194404.6 | 4197869.3 | 3.34 | 0.08 |
| *bl23* | 0.1 | 10.6 | 11.6 | 55084.5 | 56468.0 | 56527.6 | 2.51 | 0.11 |
| *bl24* | 0.1 | 5.5 | 6.6 | 46053.0 | 47843.8 | 47860.9 | 3.89 | 0.04 |
| *bs09* | 0.1 | 1.3 | 2.6 | 6189414.0 | 6237946.0 | 6248836.4 | 0.78 | 0.17 |
| *bs10* | 0.1 | 2.0 | 2.0 | 6888128.0 | 7017450.8 | 7006625.6 | 1.88 | –0.15 |
| *bs11* | 0.1 | 4.2 | 6.1 | 63413.3 | 64108.4 | 64167.3 | 1.10 | 0.09 |
| *bs12* | 0.1 | 1.5 | 2.0 | 69525.5 | 70597.5 | 70707.2 | 1.54 | 0.16 |
| *bs13* | 0.1 | 7.6 | 7.9 | 3404629.0 | 3577103.3 | 3579101.7 | 5.07 | 0.06 |
| *bs14* | 0.3 | 9.6 | 9.2 | 2618327.9 | 2826044.0 | 2821923.4 | 7.93 | –0.15 |
| *bs15* | 0.1 | 2.4 | 2.4 | 37110.0 | 38403.7 | 38392.1 | 3.49 | –0.03 |
| *bs16* | 0.2 | 3.7 | 3.9 | 29347.0 | 31010.5 | 31014.1 | 5.67 | 0.01 |
| *bs17* | 0.1 | 1.8 | 2.9 | 11336035.0 | 11367896.9 | 11373239.5 | 0.28 | 0.05 |
| *bs18* | 0.2 | 2.5 | 5.5 | 10382306.5 | 10448178.6 | 10457064.6 | 0.63 | 0.09 |
| *bs19* | 0.1 | 2.7 | 4.9 | 105449.5 | 105867.7 | 105918.2 | 0.40 | 0.05 |
| *bs20* | 0.1 | 2.8 | 4.8 | 106072.0 | 107179.7 | 107197.3 | 1.04 | 0.02 |
| *bs21* | 0.3 | 21.3 | 37.6 | 5307803.7 | 5515064.2 | 5526243.9 | 3.90 | 0.20 |
| *bs22* | 0.8 | 20.6 | 27.3 | 4180407.3 | 4418340.0 | 4423735.2 | 5.69 | 0.12 |
| *bs23* | 0.2 | 14.5 | 22.7 | 55368.0 | 57237.8 | 57328.8 | 3.38 | 0.16 |
| *bs24* | 0.2 | 10.3 | 12.1 | 48674.0 | 50761.2 | 50815.9 | 4.29 | 0.11 |
| *planar50* | 0.1 | 0.8 | 1.0 | 122199689.0 | 122218805.0 | 122218805.0 | 0.02 | 0.00 |

Table 4.5 Path decomposition computational results: use of LCIs in the root node.
N – No LCIs; S – Simple LCIs; G – General LCIs.

| *Instance* | *DD* | *DU* | *AD* | *AU* | *B* | *CD* | *CU* |
|---|---|---|---|---|---|---|---|
| *bl01* | 41 | 40 | 16 | 28 | 27 | 25 | 48 |
| *bl02* | 19 | 32 | 26 | 23 | 17 | 18 | 46 |
| *bl03* | 81 | 19 | 46 | 23 | 25 | 19 | 17 |
| *bl04* | ** | ** | ** | ** | ** | ** | ** |
| *bl05* | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| *bl06* | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| *bl07* | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| *bl08* | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| *bs01* | 3 | 4 | 3 | 3 | 3 | 3 | 3 |
| *bs02* | 11 | 9 | 9 | 5 | 11 | 5 | 7 |
| *bs03* | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| *bs04* | 289 | 125 | 202 | 159 | 102 | 120 | 191 |
| *bs05* | 13 | 20 | 31 | 18 | 14 | 14 | 19 |
| *bs06* | 42 | 40 | 24 | 32 | 34 | 28 | 33 |
| *bs07* | 135 | 111 | 107 | 73 | 199 | 142 | 125 |
| *bs08* | 8 | 12 | 20 | 17 | 23 | 23 | 23 |
| *planar30* | * | * | 2258 | 1305 | 353 | 216 | 214 |

Table 4.6 Path decomposition computational results: number of optimised nodes for different branching rules.
* Integer optimal solution not found within ten minutes.
** Feasible integer solution not found within ten minutes.

**Branching rules with RMP heuristic**

In the following tests, we introduced the use of a heuristic at the end of the optimisation of the root node. The heuristic consists in solving the RMP (with Cplex MIP Solver) of the root node, considering that all the variables must be binary. The results obtained for some of the smaller instances (the ones where the number of nodes was larger than 5 for all the branching strategies of Table 4.6) are given in Table 4.7 (columns with an H are the ones where the heuristic was used).

The improvement of the number of optimised nodes is significant for both branching rules. The proposed branching rule with heuristic gave better results concerning the number of nodes in seven instances (including the three more difficult ones) and worst in two, when compared with BH.

For instance *bl04*, none of the branching rules strategies, with or without heuristic, gave a feasible integer solution. For that instance, we tested two other search strategies: depth search and best search. Depth first was the only one to find an incumbent solution.

| *Instance* | *B* | | *CD* | | *BH* | | *CDH* | |
|---|---|---|---|---|---|---|---|---|
| | *Time* | *Nodes* | *Time* | *Nodes* | *Time* | *Nodes* | *Time* | *Nodes* |
| *bl01* | 0.7 | 27 | 0.8 | 25 | 0.7 | 17 | 0.7 | 20 |
| *bl02* | 10.8 | 17 | 11.1 | 18 | 1.7 | 11 | 1.7 | 8 |
| *bl03* | 6.3 | 25 | 4.8 | 19 | 8.9 | 20 | 8.0 | 19 |
| *bl04* | ** | ** | ** | ** | ** | ** | ** | ** |
| *bs02* | 1.7 | 11 | 1.0 | 5 | 2.3 | 12 | 1.6 | 6 |
| *bs04* | 65.2 | 102 | 85.3 | 120 | 71.1 | 75 | 53.8 | 73 |
| *bs05* | 1.3 | 14 | 1.3 | 14 | 1.6 | 14 | 1.5 | 14 |
| *bs06* | 1.2 | 34 | 0.9 | 28 | 1.4 | 23 | 1.6 | 25 |
| *bs07* | 10.3 | 199 | 7.5 | 142 | 5.8 | 98 | 4.6 | 78 |
| *bs08* | 0.6 | 23 | 0.6 | 23 | 0.8 | 24 | 0.8 | 23 |
| *planar30* | 34.7 | 353 | 17.4 | 216 | 10.5 | 144 | 6.8 | 73 |

Table 4.7 Path decomposition computational results for branching rules with the RMP heuristic. ** Feasible integer solution not found within one hour.

For the larger instances, we used depth search and the RMP heuristic. In addition, we removed columns with reduced cost greater than the gap, in every five iterations. The results for these instances are given in Table 4.8 for the (Barnhart et al., 2000) branching rule and the one proposed. The results were similar in almost all instances. The time results were significantly different in three instances (*bl11*, *bl14*, and *bl15*), with the proposed branching rule giving better results in all of those. The quality of the incumbent solutions was different in two instances (*bl12* and *bl22*), in favour of the (Barnhart et al., 2000) branching rule in one instance and in favour of the proposed branching rule in the other.

### 4.5.4 Comparative computational results

#### Implementation issues for the knapsack decomposition

For the knapsack decomposition we used the COIN implementation of the Horowitz-Sahni algorithm (Horowitz and Sahni, 1974) to solve the knapsack (sub)problems. We used the Cplex 8.1 dual simplex algorithm to solve the RMPs. The branching rule proposed in subsection 4.3.3 (considering one commodity and a set of arcs leaving the same node) with down branching first was used. We kept the default search strategy of *ADDing*: using a depth first search when the node generates descendants and best bound search otherwise.

| *Instance* | *BH* | | *CDH* | |
|---|---|---|---|---|
| | *Time* | *Value* | *Time* | *Value* |
| *bl09* | ** | ** | ** | ** |
| *bl10* | ** | ** | ** | ** |
| *bl11* | 725.0 | 69018 | 470.8 | 69018 |
| *bl12* | * | 66022 | * | 66019 |
| *bl13* | * | 3155673 | * | 3155673 |
| *bl14* | 371.7 | 2433011 | 303.6 | 2433011 |
| *bl15* | 359.1 | 34274 | 318.5 | 34274 |
| *bl16* | 38.6 | 28074 | 39.1 | 28074 |
| *bl17* | * | 13324233 | * | 13324233 |
| *bl18* | ** | ** | ** | ** |
| *bl19* | ** | ** | ** | ** |
| *bl20* | ** | ** | ** | ** |
| *bl21* | * | 5837994 | * | 5837994 |
| *bl22* | * | 4216651 | * | 4217172 |
| *bl23* | * | 56970 | * | 56970 |
| *bl24* | * | 51081 | * | 51081 |
| *bs09* | * | 6308373 | * | 6308373 |
| *bs10* | ** | ** | ** | ** |
| *bs11* | ** | ** | ** | ** |
| *bs12* | ** | ** | ** | ** |
| *bs13* | * | 3615375 | * | 3615375 |
| *bs14* | * | 3181860 | * | 3181860 |
| *bs15* | 115.2 | 38533 | 120.7 | 38533 |
| *bs16* | 344.0 | 31124 | 337.8 | 31124 |
| *bs17* | ** | ** | ** | ** |
| *bs18* | ** | ** | ** | ** |
| *bs19* | * | 106369 | * | 106369 |
| *bs20* | ** | ** | ** | ** |
| *bs21* | ** | ** | ** | ** |
| *bs22* | * | 4796079 | * | 4796079 |
| *bs23* | * | 57821 | * | 57821 |
| *bs24* | * | 51045 | * | 51045 |
| *planar50* | * | 123226335 | * | 123226335 |

Table 4.8 Path decomposition computational results: two branching rules in the larger instances.
* Integer optimal solution not found within one hour.
** Feasible integer solution not found within one hour.

The first RMP is constructed based on the original variables. Each original variable corresponds to one arc *ij* and one commodity *k*. If the demand of commodity *k* is lower than or equal to the capacity of arc *ij*, then this variable is used on the RMP, since it corresponds to the extreme point of the subproblem in which commodity *k* is the only one with value equal to one, that is, the commodity *k* is the only one to traverse arc *ij*. In the first RMP only the flow conservation constraints related with the origin constraints are present. When that relaxation is solved (that is, the subproblem does not generate any attractive column), constraints that are not present in the RMP are checked and, in the case they are violated by the current solution, associated columns are inserted in the RMP. We denote this strategy by dynamic row management.

For the smaller instances the usual model was used, that is, all flow conservation constraints are equalities.

For the larger instances we used the decomposition based on the model with inequality constraints *(MOB)* presented in Section 4.2 (page 96), the inexact RMP strategy and removal of columns in some iterations, both described in subsection 4.4.2 (page 108).

We used the dynamic management of rows (also described in subsection 4.4.2, page 108) for all the instances, since we soon realised that it is a crucial factor for the efficiency of this decomposition. For example, the root node of instance *bl01* took *873.5* seconds to be solved without dynamic management of rows, as opposed to the *8.4* seconds to be showed in Table 4.9.

**Smaller instances**

Table 4.9 presents the results for the three approaches (columns *P* for path decomposition, columns *K* for the knapsack decomposition and column *O* for the original formulation solved with Cplex) for the smaller instances.

The “Root / LR gap” column refers to the relative percentage gap between the optimal integer value and the value of the root node for the two decompositions (including the use of LCIs in the case of the path decomposition), and between the optimal integer value and the value of the linear relaxation in the case of original formulation solved by Cplex.

The “Nodes” and “Time” columns refer to the number of optimised nodes for the three approaches and the total optimisation time, respectively.

The quality of the lower bound of the root node given by the knapsack decomposition is significantly better than the one provided by the linear relaxation of original formulation or even by the root node of the path decomposition with the use of LCIs. With two exceptions, it is always less than 0.5%. For instance *bl04* it is larger than 0.5%, but significantly smaller than any one of the other approaches. The quality of the lower bounds provided by the knapsack decomposition justifies the smaller number of nodes that were optimised for all instances, when

compared with the path decomposition (for five instances, the optimal solution was found in the root node). When the number of nodes is compared with Cplex, we must note that the linear relaxation value of that approach is not the same as its root value. Cplex uses several different families of cuts, namely in the root node, when solving an integer problem. That may explain the fact that the number of optimised nodes of Cplex is, for some instances, smaller than the one of the knapsack decomposition.

| *Instance* | *IP Value* | *Root / LR gap* | | | *Nodes* | | | *Time* | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | *P* | *K* | *O* | *P* | *K* | *O* | *P* | *K* | *O* |
| *bl01* | 1615947 | 0.4 | 0.2 | 4.1 | 20 | 13 | 1 | 0.7 | 8.3 | 0.6 |
| *bl02* | 1816947 | 1.3 | 0.0 | 9.6 | 8 | 1 | 1 | 1.7 | 13.1 | 1.5 |
| *bl03* | 17340 | 0.5 | 0.0 | 7.9 | 19 | 1 | 32 | 8.0 | 4.8 | 4.6 |
| *bl04* | 21370 | 10.8 | 6.3 | 16.4 | ** | 112 | 567 | ** | 1286.8 | 54.6 |
| *bl05* | 474782 | 0.0 | 0.0 | 1.0 | 1 | 1 | 1 | 0.1 | 7.6 | 0.3 |
| *bl06* | 411480 | 0.0 | 0.0 | 1.0 | 1 | 1 | 1 | 0.0 | 1.7 | 0.3 |
| *bl07* | 5751 | 0.0 | 0.0 | 0.6 | 1 | 1 | 1 | 0.0 | 3.1 | 0.3 |
| *bl08* | 5688 | 0.0 | 0.0 | 0.5 | 1 | 1 | 1 | 0.0 | 1.3 | 0.3 |
| *bs01* | 1639862 | 0.0 | 0.0 | 6.2 | 3 | 1 | 1 | 0.3 | 1.4 | 0.1 |
| *bs02* | 1702368 | 0.5 | 0.0 | 8.6 | 6 | 1 | 5 | 1.6 | 6.0 | 1.9 |
| *bs03* | 16828 | 0.0 | 0.0 | 1.4 | 1 | 1 | 1 | 0.1 | 2.8 | 0.1 |
| *bs04* | 20213 | 3.5 | 0.2 | 8.7 | 73 | 5 | 3 | 53.8 | 49.7 | 2.0 |
| *bs05* | 500870 | 0.3 | 0.0 | 8.3 | 14 | 5 | 1 | 1.5 | 2.7 | 2.4 |
| *bs06* | 502151 | 0.8 | 0.4 | 6.9 | 25 | 18 | 3 | 1.6 | 4.9 | 3.7 |
| *bs07* | 7223 | 1.1 | 0.5 | 9.5 | 78 | 50 | 75 | 4.6 | 9.1 | 8.1 |
| *bs08* | 6471 | 0.7 | 0.0 | 5.2 | 23 | 1 | 1 | 0.8 | 1.8 | 0.8 |
| *planar30* | 44471934 | 0.0 | *** | 0.3 | 73 | *** | 8 | 6.8 | *** | 2.0 |

Table 4.9 Computational results: smaller instances.
** Feasible integer solution not found within one hour.
*** Root node not solved within one hour.

In general, the quality of the lower bounds of the knapsack decomposition does not translate into a competitive solution time, when compared with the other two approaches (with some exceptions when compared with the path decomposition, as, for example, in the instance with greater gap, *bl04*, in which the path decomposition did not obtain an optimal solution in one hour, and the knapsack decomposition obtained it in 20 minutes, approximately). The knapsack decomposition could not solve the root node of the *planar30* instance in one hour, while the path decomposition and Cplex obtained an optimal binary solution in a few seconds.

The path decomposition gave better time results than Cplex for the instances with a larger number of arcs (*bl05* to *bl08* and *bs05* to *bs08*). For the other instances, it gave worst time results, in particular for instance *bl04*, where an optimal solution was not obtained within one hour.

Table 4.10 gives detailed results for the instances *bl04* and *bs04* for both decompositions. A significant difference between the two is the time spent on solving the root node. For the path decomposition, that time is negligible, while for the knapsack decomposition the same does not happen (which is a major issue in the large instances to be analysed next). Solving the RMPs is the main time-consuming task for both decompositions. For the knapsack decomposition, although the binary knapsack subproblems are not solvable in polynomial time, the time spent on them represents a very small percentage of the overall time. The same happens with the generation of LCIs for the path decomposition.

**Larger instances**

In Table 4.11, results are presented for the larger instances. Columns "RQ" give the relative increase of the IP solution obtained by the two decompositions when compared with the one from the original formulation (an equal sign means that the optimal solution was obtained).

The results are surprising. The general-purpose solver Cplex 8.1 obtained an optimal solution in *23* out the *33* instances, while the path decomposition achieved it in only *6* instances. For the others, although the root node was solved in a small amount of time, the search of the tree was ineffective. Only on the instances with a large number of arcs and a medium number of commodities (*bl13* to *bl16* and *bs13* to *bs16*) the path decomposition gave time results that approximate the ones of Cplex.

Knapsack decomposition gave an optimal solution in only *4* instances and could not solve the root node in *24*. For the *7* instances where the root node was solved, the comparative quality of lower bound given by this decomposition was confirmed.

To our best knowledge, a computational study comparing a general-purpose solver with a specific method for the binary MFP was never done before. Quoting (Barnhart et al., 2000), "Without decomposition, these LP relaxations [original formulation linear relaxation and linear relaxation of the (full) master of the path formulation] may require excessive memory and/or run times to solve."

The computational results here presented suggest precisely the opposite. Even improving the (Barnhart et al., 2000) approach (with general LCIs, a RMP heuristic and a slightly better branching rule, judging from the smaller instances results), Cplex 8.1 clearly outperforms the branch-and-price-and-cut specific method.

The coding efficiency of Cplex 8.1 can be easily confirmed by the large number of nodes it searches, compared with the much smaller number of the decompositions (for example, in

instance *bl17* Cplex searched *8807* nodes of the tree, while the path decomposition only *321* in same amount of time).

Furthermore, Cplex uses several different families of cuts (for example, clique, cover, flow cover, disjunctive, Gomory, ...), heuristics in the nodes of the tree, strong branching, preprocessing, just to name a few of the issues involved in that state-of-the-art implementation of branch-and-cut.

A point worth noting is that the same approach used in the proposed branching rule can be used with all types of constraints that are based on the original variables, namely with the families of cuts Cplex uses. The proposed branching rule demonstrates how both formulations can be used together: all the constraints (cuts or branches) that are derived based on the original variables can be used in the path decomposition in the same manner as the proposed branching rule and the LCIs were used, that is, by keeping them in the RMP. Although this approach is nothing more than dualising the constraints that do not define the subproblems in all of the problems of the nodes of the tree, it allows a generalisation that is not usually explored in branch-and-price-and-cut algorithms, where the branching is forced in the subproblem and cuts must be dealt with some specific procedure.

| *Instance* | | *bl04* | | *bs04* | |
|---|---|---|---|---|---|
| | | *P* | *K* | *P* | *K* |
| *Total* | *Time* | 605.8 | 1286.8 | 53.8 | 49.7 |
| ***Root node*** | *Time* | 0.3 | 25.1 | 0.1 | 27.5 |
| | *Iterations* | 71 | 333 | 53 | 427 |
| | *RMP time (%)* | 73.3 | 90.4 | 43.6 | 91.1 |
| | *SP time (%)* | 13.2 | 2.6 | 0.0 | 2.2 |
| | *LCIs time (%)* | 36.7 | – | 31.6 | – |
| ***Other nodes*** | *Time* | 601.5 | 1259.4 | 52.2 | 22.2 |
| | *Nodes* | 65 | 113 | 73 | 5 |
| | *Time/Node* | 9.3 | 11.1 | 0.7 | 4.4 |
| | *Iterations* | 829 | 2692 | 574 | 253 |
| | *RMP time (%)* | 96.4 | 97.5 | 93.1 | 92.5 |
| | *SP time (%)* | 0.9 | 1.1 | 1.6 | 2.1 |
| | *LCIs time (%)* | 0.5 | – | 1.6 | – |

Table 4.10 Detailed computational results for two instances for the decompositions.
– Does not apply.

| *Instance* | *IP value* | *Root / LR gap* | | | *RQ* | | *Nodes* | | | *Time* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *P* | *K* | *O* | *P* | *K* | *P* | *K* | *O* | *P* | *K* | *O* |
| *bl09* | 6261671.0 | 1.1 | *** | 2.5 | ** | *** | ** | *** | 4155 | ** | *** | 1087.2 |
| *bl10* | ** | ** | *** | ** | ** | *** | ** | *** | ** | ** | *** | ** |
| *bl11* | 69018.0 | 0.1 | *** | 1.0 | = | *** | 774 | *** | 234 | 470.8 | *** | 17.5 |
| *bl12* | 65902.0 | 0.4 | *** | 1.7 | 0.2 | *** | * | *** | 614 | * | *** | 182.1 |
| *bl13* | 3132695.0 | 0.3 | 0.0 | 3.9 | 0.7 | = | * | 16 | 276 | * | 1766.6 | 176.5 |
| *bl14* | 2433011.0 | 0.4 | 0.1 | 4.4 | = | 0.2 | 835 | * | 108 | 303.6 | * | 117.9 |
| *bl15* | 34274.0 | 0.2 | 0.0 | 2.9 | = | 0.0 | 283 | * | 289 | 318.5 | * | 96.3 |
| *bl16* | 28074.0 | 0.2 | 0.1 | 4.4 | = | = | 117 | 31 | 62 | 39.1 | 1359.4 | 57.1 |
| *bl17* | 13190922.0 | 0.3 | *** | 0.8 | 1.0 | *** | * | *** | 8807 | * | *** | 2881.7 |
| *bl18* | 10496120.0 | 0.3 | *** | 0.9 | ** | *** | ** | *** | 3928 | ** | *** | 1448.4 |
| *bl19* | * 109556.0 | 0.9 | *** | 1.4 | ** | *** | ** | *** | * | ** | *** | * |
| *bl20* | * 111604.0 | 0.9 | *** | 1.8 | ** | *** | ** | *** | * | ** | *** | * |
| *bl21* | * 5800149.0 | 0.5 | *** | 3.2 | 0.7 | *** | * | *** | * | * | *** | * |
| *bl22* | 4209266.0 | 0.3 | *** | 3.6 | 0.2 | *** | * | *** | 144 | * | *** | 389.1 |
| *bl23* | 56856.0 | 0.6 | *** | 3.1 | 0.2 | *** | * | *** | 2176 | * | *** | 2250.2 |
| *bl24* | 47964.0 | 0.2 | *** | 4.0 | 6.5 | *** | * | *** | 234 | * | *** | 250.8 |
| *bs09* | 6287195.0 | 0.6 | *** | 1.6 | 0.3 | *** | * | *** | 3238 | * | *** | 665.3 |
| *bs10* | 7072735.0 | 0.9 | *** | 2.6 | ** | *** | ** | *** | 3363 | ** | *** | 643.2 |
| *bs11* | * 65168.0 | 1.5 | *** | 2.7 | ** | *** | ** | *** | * | ** | *** | * |
| *bs12* | 71483.0 | 1.1 | *** | 2.7 | ** | *** | ** | *** | 4811 | ** | *** | 2235.8 |
| *bs13* | 3605397.0 | 0.7 | 0.3 | 5.6 | 0.3 | 1.6 | * | * | 579 | * | * | 532.2 |
| *bs14* | * 2872664.0 | 1.8 | 1.0 | 8.9 | 10.8 | ** | * | ** | 2206 | * | ** | * |
| *bs15* | 38533.0 | 0.4 | 0.1 | 3.7 | = | = | 131 | 37 | 133 | 120.7 | 290.2 | 99.0 |
| *bs16* | 31124.0 | 0.4 | 0.0 | 5.7 | = | = | 259 | 1 | 45 | 337.8 | 630.0 | 95.3 |
| *bs17* | * 11447995.0 | 0.7 | *** | 1.0 | ** | *** | ** | *** | * | ** | *** | * |
| *bs18* | 10486796.0 | 0.3 | *** | 1.0 | ** | *** | ** | *** | 1719 | ** | *** | 746.2 |
| *bs19* | 106142.0 | 0.2 | *** | 0.7 | 0.2 | *** | * | *** | 7218 | * | *** | 913.0 |
| *bs20* | 107712.0 | 0.5 | *** | 1.5 | ** | *** | ** | *** | 2142 | ** | *** | 896.1 |
| *bs21* | * 5562469.0 | 0.7 | *** | 4.6 | ** | *** | ** | *** | * | ** | *** | * |
| *bs22* | * 4487045.0 | 1.4 | *** | 6.8 | 6.9 | *** | * | *** | * | * | *** | * |
| *bs23* | 57548.0 | 0.4 | 0.1 | 3.8 | 0.5 | ** | * | ** | 2547 | * | ** | 3301.3 |
| *bs24* | 50980.0 | 0.3 | *** | 4.5 | 0.1 | *** | * | *** | 1555 | * | *** | 2919.6 |
| *planar50* | * 122272951.0 | 0.0 | *** | 0.1 | 0.8 | *** | * | *** | * | * | *** | * |

Table 4.11 Computational results: larger instances.
* Integer optimal solution not found within one hour.The IP value is the value of the incumbent solution given by Cplex.
** Feasible integer solution not found within one hour.
*** Root node not solved within one hour.
= Optimal solution.

## 4.6 Conclusions

In this Chapter, we presented two decompositions, and developed two branch-and-price algorithms in order to solve exactly the binary multicommodity flow problem. The branch-and-price-and-cut algorithm for the path decomposition follows the approach presented in (Barnhart et al., 2000). We extend the approach given in that reference by using general LCIs instead of simple LCIs and a new RMP heuristic. Our main contribution, with respect to that decomposition, was the development of a new branching rule that gave slightly better results for the smaller instances tested than the one proposed in that reference. In addition, the new branching rule suggests how other families of cuts can be used in a more effective branch-and-price-and-cut algorithm.

The second branch-and-price algorithm is based on a knapsack decomposition, which allows obtaining better lower bounds in the nodes of the branch-and-price tree. The quality of the lower bounds was proved empirically by the computational tests performed. However, it turned out that, in particular for the larger instances tested, this decomposition is particularly difficult to deal with. Even with dynamic management of rows and columns, and with an inexact RMP strategy to speed up its solution time, the larger instances could not be solved in a reasonable amount of time. Column generation stabilisation, analytic center cutting plane or bundle methods are natural candidates for the improvement of the efficiency of this decomposition approach.

Solving the original formulation with Cplex 8.1 clearly outperformed the two decomposition approaches presented here. The exception was a set of few instances with a large number of arcs, where the path decomposition gave similar results. To our best knowledge, the comparison between decomposition approaches and a general-purpose solver for the binary MFP was made here for the first time. Given the fact that, in the literature, that approach is taken as non promising, this result came as a surprise.

# 5 Accelerating Column Generation for Planar Multicommodity Flow Problems

In this Chapter, we present a way of accelerating a column generation algorithm for the linear minimum cost multicommodity flow problem. We use a new model that, besides the usual variables corresponding to flows on paths, has a polynomial number of extra variables (when the problem is defined in a planar network), corresponding to flows on circuits. Those extra variables are explicitly considered in the restricted master problem, from the beginning of the column generation process. The subproblem remains a set of shortest path problems, one for each commodity.

We present computational results for the comparison of this new approach with standard column generation, a bundle method, and a general-purpose solver. For the tested instances, there is an effective improvement in computational time of the column generation method when the model with extra variables is used.

## 5.1 Introduction

In this Chapter we present an approach for accelerating column generation for the linear minimum cost multicommodity flow problem (MFP).

Column generation is a technique to solve linear programs with a large (possibly exponential) number of variables. One important group of these large linear programs is the one resulting from applying a Dantzig-Wolfe decomposition (DWD).

Although column generation has been known for several decades (its roots lie in the work of Ford and Fulkerson (Ford and Fulkerson, 1958), Dantzig and Wolfe (Dantzig and Wolfe, 1960), and Gilmore and Gomory (Gilmore and Gomory, 1961; Gilmore and Gomory, 1963)), a renewed interest about this subject can be noted in the last few years. Reasons for that may be found in the evolution of the hardware that made possible to solve large practical problems and in the availability of robust and efficient commercial software to solve linear programs (Bixby et al., 2000). In this way, memory requirements and coding complexity, which could be thought as major disadvantages when implementing a column generation algorithm (when compared with a subgradient algorithm to solve its dual, for example), became less problematic.

Another reason for the renewed interest in column generation comes from the better understanding of its potential to solve integer problems. Branch-and-price algorithms (combining column generation and branch-and-bound) have been also a major topic of research in the last few years, having its root in the work of Desrosiers, Soumis and Desrochers (Desrosiers et al., 1984). Surveys on the column generation / branch-and-price methods can be found in (Soumis, 1997; Barnhart et al., 1998; Wilhelm, 2001; Lübbecke and Desrosiers, 2002).

One major disadvantage of column generation is the well-known tail-off effect that, usually, implies a slow convergence of the method. To deal with that issue several methods have been devised (see (Lübbecke and Desrosiers, 2002) for a more detailed description and additional references), usually taking a dual perspective, given that column generation can be seen as a cutting plane algorithm applied to the dual problem.

Marsten et al. (Marsten et al., 1975) developed the boxstep method, in which the dual variables, in each iteration, are confined to lie inside or in the boundary of a box centred in the previous solution; Wentges (Wentges, 1997) suggested the use of a convex combination between the best dual variables found so far and the optimal dual variables of the current iteration; du Merle et al. (Merle et al., 1999), introduced a stabilisation method that amounts to penalising solutions that lie outside a predetermined box, which may be adjusted as the algorihtm proceeds, and includes a right-hand side perturbation; Kallehauge, Larsen, and

Madsen (Kallehauge et al., 2001) proposed a trust region method to try to avoid the oscillation of the dual variables by restricting them to a box with an automatically adjusted size; finally, in bundle methods (Frangioni, 2002), a penalisation is added to points that are distant from the current one, which implies a quadratic term in the objective function of the model, leading to a nonlinear master problem – for a clear explanation of the relation between bundle methods and column generation / cutting plane methods, see (Frangioni, 1997).

Our present work is based on the method proposed by Valério de Carvalho (Carvalho, 2000). The author presented a way of reducing the tail-off effect by including a polynomial number of extra dual optimal cuts (extra primal variables) in the restricted master problem prior to the beginning of the optimisation process. The motivation is the following: the use of a tighter dual space restriction from the start may help in finding the optimal solution faster. Under certain conditions, the primal space is not relaxed, and it is possible to recover an optimal solution to the original problem from the optimal solution to the extended model. In this work we apply a similar idea to the linear minimum cost MFP defined over planar graphs.

The minimum cost MFP is defined over a network in which we want to route, with minimal cost, a set of commodities from their origins to their destinations without exceeding the capacities of the arcs.

This problem, as well as related multicommodity flow problems, has been the subject of interest of the research community for its applications (namely in transportation/distribution systems, telecommunications networks and production planning) and for its role as a typical model with the so-called block-angular structure, thus being a representative problem (maybe the most used one) for which several decomposition methods can be applied and tested.

A description of several applications can be found in the surveys presented in (Assad, 1978; Kennington, 1978; Ahuja et al., 1993). In the same surveys, classical methods (since they were first developed in the 1960s and 1970s) are described (which can be grouped in price directive decompositions, basis partitioning methods, and resource-directive decompositions). More recently, several approaches have been developed. Among them, we refer to specialised interior point methods (Schultz and Meyer, 1991; Castro, 2000) and bundle methods (Frangioni and Gallo, 1999). A more detailed review on solution methods for the linear MFP was given in Chapter 3, subsection 3.2.4 (page 65).

Our present work deals with the origin-destination MFP, that is, problems where a commodity is defined by the node in which it is supplied and by the node in which it is required. However, it can be easily extended to problems with multiple origins and destinations, using the more general framework presented in Chapter 3 in the context of the integer MFP.

We consider that, for each arc, the costs of all commodities are the same. This happens in

some applications of the multicommodity flow models, as, for example, in computer networks, where the commodities are usually associated with streams of traffic between different pairs of users.

As we said before, our present work is based on the approach first presented in (Carvalho, 2000). The basic idea is that by adding extra variables (extra dual cuts), prior to the beginning of the column generation process, we can speed up the column generation algorithm and improve its practical convergence properties. The results reported in (Carvalho, 2000) for the cutting stock problem show that, for some classes of instances, the speed up factor is close to five.

The extra variables added to the model must be tailored to the specific problem at hand. In the case of the MFP, we use variables associated with flows on circuits. An important issue is that the number of extra variables must be tractable for the master problem. Given that the number of circuits of a general network is exponential with respect to its size, we restrict our present work to the MFP defined on planar graphs, where it is possible to select a polynomial number of elementary circuits, based on which all circuits of the network can be implicitly considered.

This Chapter is organised as follows. In Section 5.2, we resume the original formulation and the formulation that results from applying DWD, giving origin to the path decomposition formulation. In Section 5.3, we briefly describe the standard column generation procedure for the problem in study. In Section 5.4, we present and discuss our approach to accelerate column generation when applied to the MFP. In Section 5.5, we prove that the number of extra variables used is polynomial with respect to the dimension of the network. In Section 5.6, we describe and discuss the results of the computational tests performed. In Section 5.7, we present our main conclusions and future work directions.

## 5.2 Formulations

### 5.2.1 Arc formulation

We consider a network formed by a set of $n$ nodes, represented by $N$, and a set of $m$ arcs, represented by $A$. We use an index $i=\{1,...,n\}$ to represent a node and a pair of indices $ij$ to represent an arc which has origin in node $i$ and destination in node $j$. We define a set $K$ of $h$ commodities, indexed by $k$. Each commodity $k$ is characterised by an origin, $o^k$, a destination, $d^k$, and an integer demand, $r^k$, which is the number of units that are supplied at its origin and that are required at its destination. We also define a capacity, $u_{ij}$, and a linear unit cost, $c_{ij}$, both associated with each arc of the network. We make the usual assumption $c_{ij} \geq 0, \ \forall ij \in A$.

The arc formulation of the minimum cost MFP is obtained using decision variables that

represent the flows in all arcs for all commodities. Those decision variables are represented as $x_{ij}^k$. The formulation is as follows:

$$Min \sum_{ij \in A} \sum_{k \in K} c_{ij} x_{ij}^k$$

*subject to:*

$$\sum_{j:ij \in A} x_{ij}^k - \sum_{j:ji \in A} x_{ji}^k = b_i^k, \ \forall i \in N, \ \forall k \in K \qquad (5.1)$$

$$\sum_{k \in K} x_{ij}^k \leq u_{ij}, \ \forall ij \in A \qquad (5.2)$$

$$x_{ij}^k \geq 0, \ \forall ij \in A, \ \forall k \in K,$$

where

$$b_i^k = \begin{cases} r_k \ if \ i = o^k \\ -r_k \ if \ i = d^k \\ 0 \ otherwise. \end{cases}$$

Constraints *(5.1)* are flow conservation constraints. They state that, for each commodity, the difference between the flow that enters a node and the flow that leaves that node is equal to the supply/demand of that node.

Constraints *(5.2)* are capacity constraints. They state that the total flow on each arc must be less than or equal to its capacity. Without these constraints, an optimal solution to the problem could be found by solving independent shortest path problems (one for each commodity).

### 5.2.2 Path formulation

We represent the set of all simple paths between the origin and the destination of commodity $k$ by $P^k$. If the arc $ij$ belongs to path $p$ of commodity $k$, then $y_{ij}^{pk}$ equals *1*; otherwise, $y_{ij}^{pk}$ equals *0*. The unit flow cost of path $p$ of commodity $k$, is $c^{pk} = \sum_{ij \in A} y_{ij}^{pk} c_{ij}, \ \forall p \in P^k, \ \forall k \in K$.

When we refer to a path of a commodity, we mean a simple path that begins at the origin of a commodity and ends at its destination.

The path formulation is obtained by defining the decision variables, $\lambda^{pk}$, as the flow on each path of all commodities.

$$Min \sum_{k \in K} \sum_{p \in P^k} c^{pk} \lambda^{pk}$$

*subject to:*

$$\sum_{p \in P^k} \lambda^{pk} = r^k, \ \forall k \in K \qquad (5.3)$$

$$\sum_{k \in K} \sum_{p \in P^k} y_{ij}^{pk} \lambda^{pk} \leq u_{ij}, \ \forall ij \in A \qquad (5.4)$$

$$\lambda^{pk} \geq 0, \ \forall k \in K, \ \forall p \in P^k.$$

This formulation can be obtained directly by defining decision variables, $\lambda^{pk}$, that represent the flow on path $p$ of commodity $k$.

Constraints *(5.3)* ensure that the demand is routed from the origin to the destination, for all commodities, and constraints *(5.4)* are the capacity constraints.

As proved in (Tomlin, 1966), the path formulation can be obtained by applying a DWD on the arc formulation. In fact, the roots of DWD can be found in the approach of Ford and Fulkerson (Ford and Fulkerson, 1958) to the maximal multicommodity flow problem as stated in (Dantzig, 1963).

Although being equivalent, the practical behaviour of the two just presented formulations can be very different. The arc formulation has $m.h$ variables and $n.h+m$ constraints, while the path formulation has only $h+m$ constraints and an exponential number of variables (with respect to the size of the network). However, $h+m$ is an upper bound for the number of variables with a positive value in a basic solution.

Since the size of the basis is one major factor to simplex methods' efficiency, we can expect the path formulation to be more efficient in larger instances, given that we use a column generation scheme to deal with the huge number of columns (even for moderate size instances).

We note that it is always possible to find a feasible solution to a model from a feasible solution to the other model with the same cost. The ways of performing those transformations and their proofs can be found in (Ahuja et al., 1993).

We finish this Section by noting that there are other possible decompositions leading to different problems, namely by the aggregation of commodities by origin or destination. However, computational experiments (and the fact that it seems easier for the master problem to combine "smaller pieces") point to the one used here as being the most efficient (Jones et al., 1993).

## 5.3 Standard Column Generation

### 5.3.1 Overview

In the resolution of the path formulation using column generation, a restricted master problem (RMP), that is a problem where not all paths are considered, is optimised first. After optimising the RMP, the attractiveness of the paths that are not present in the RMP is evaluated by solving a subproblem that uses the values of the dual variables. That subproblem consists in determining the shortest path between their origin and their destination for all commodities in a network with modified costs. After inserting the attractive paths in the RMP, the procedure is repeated until no more attractive paths are returned by the subproblem. A detailed exposition of this procedure is presented in (Ahuja et al., 1993). Here we just note that a path $p$ of a commodity $k$ is attractive if

$$\sum_{ij \in A} y_{ij}^{pk}(w_{ij} + c_{ij}) < \pi^k \qquad (5.5)$$

where $w_{ij} \geq 0, \ \forall ij \in A$, is the dual variable associated with the constraints *(5.4)* and $\pi^k, \ k \in K$, is the dual variable (unrestricted in sign) associated with constraints *(5.3)*. Inequality *(5.5)* justifies the fact that the subproblem of each commodity can be solved by a shortest path subproblem in a network with modified costs.

### 5.3.2 Implementation issues

One critical issue must be decided when implementing a column generation algorithm, namely the way of obtaining the first RMP. The first RMP must be feasible and, in general, it is not clear how to select a set of columns that ensures that. Furthermore, the set of columns that is chosen to be part of the first RMP, even when they do not guarantee a feasible solution (thus used in conjunction with artificial variables), may be decisive in the efficiency of the method. We used an algorithm that is based on solving shortest path problems of each commodity (its associated columns being inserted in the first RMP), successively reducing the available capacity of the arcs. When it is not possible to send the flow (or part of it) of a commodity from its origin to its destination, an artificial variable is inserted in order to satisfy the flow conservation constraint associated with it. Comparative computational tests with other strategies can be found in Chapter 3.4.3 (page 76).

Another important issue is the column management. Our previous experience is that when the number of variables is manageable by the LP solver, it is better to keep all the columns generated always in the RMP.

## 5.4 Accelerating Column Generation

### 5.4.1 An extended model with circuits

The main idea behind our procedure to accelerate the column generation method to solve the linear MFP is to add variables corresponding to circuits to the path formulation, thus obtaining a new model. In this subsection we present that extended model. In subsection 5.4.3 we will show its equivalence with the original one. In subsection 5.4.6 a small example is given.

The circuits that we consider have no arc repetitions, are oriented and formed by, at least, three arcs. One of the arcs of the circuit is traversed in the opposite direction of its orientation (backward arc) and all the remaining arcs are traversed in the same direction as their orientation (forward arcs).

We consider the set $D$ formed by all the circuits such as the ones defined above, to be indexed by $s=1,...,|D|$. From now on, when referring to circuits, we mean circuits belonging to $D$, except when explicitly stated otherwise.

We define a parameter $\gamma_{ij}^s$, $\forall ij \in A$, $\forall s \in D$, that is equal to $1$, if arc $ij$ is a forward arc of circuit $s$; equal to $-1$ if arc $ij$ is the backward arc of circuit $s$; equal to $0$, if arc $ij$ does not belong to circuit $s$. We associate with each circuit $s$, $s \in D$, a variable $d^s \geq 0$, which corresponds to the flow in circuit $s$. The unit flow cost of circuit $s$ is $c^s = \sum_{ij \in A} \gamma_{ij}^s \, c_{ij}$, $\forall s \in D$.

By using the circuits defined above, it may be possible to represent one solution of the path model by a set of paths and circuits with flow, since some paths with flow can be represented as the sum of other paths and circuits. We note that the definition of the unitary costs of the paths and of the circuits implies that the cost of the solution is the same in both representations.

**Example 5.1**

Consider a (partial) solution in which there is a flow of commodity $k$ in the two paths represented in Figure 5.1.

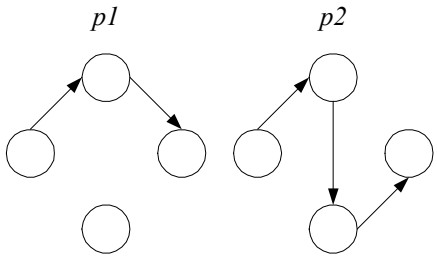



Figure 5.1 One solution represented with two paths.

For easiness of explanation, suppose that $\lambda^{p1k} \geq \lambda^{p2k}$. The same (partial) solution can be represented as showed in Figure 5.2, where the flow on path *p1*, $\lambda_{p1}^{k*}$, is now $\lambda_{p1}^{k*} = \lambda_{p1}^{k} + \lambda_{p2}^{k}$ and $d^s = \lambda^{p2k}$, since $p1 + s = p2$.

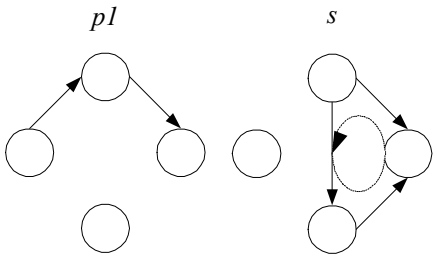



Figure 5.2 One solution represented as one path and one circuit.

♦

We now present an extended model that, besides the path variables, also includes circuit variables.

$$Min \sum_{k \in K} \sum_{p \in P^k} c^{pk} \lambda^{pk} + \sum_{s \in D} c^s d^s$$

*subject to:*

$$\sum_{p \in P^k} \lambda^{pk} = r^k, \ \forall k \in K \qquad (5.6)$$

$$\sum_{k \in K} \sum_{p \in P^k} y_{ij}^{pk} \lambda^{pk} + \sum_{s \in D} \gamma_{ij}^s d^s \geq 0, \ \forall ij \in A \qquad (5.7)$$

$$\sum_{k \in K} \sum_{p \in P^k} y_{ij}^{pk} \lambda^{pk} + \sum_{s \in D} \gamma_{ij}^s d^s \leq u_{ij}, \ \forall ij \in A \qquad (5.8)$$

$$\lambda^{pk} \geq 0, \ \forall k \in K, \ \forall p \in P^k$$

$$d^s \geq 0, \ \forall s \in D.$$

We can see this model as giving an optimal solution on paths (to satisfy constraints *(5.6)*) and then, through the circuits, redirecting their flow to other paths (that, in the case of a RMP, may not be explicitly present in the model).

Circuit variables are not present in constraints *(5.6)*, since they respect the flow conservation of all nodes.

We do not associate commodities with the circuit variables because they all have the same cost and the circuits only have one backward arc. Thus, the choice of the commodity for which its flow (or part of it) is redirected is irrelevant. In this way, we keep the number of new variables and constraints small. Furthermore, *(5.7)* and *(5.8)* can be joined to form a ranged constraint.

The new set of constraints *(5.7)* states that it is only possible to redirect flow through a

circuit if that flow exists. If we had not constrained the total flow of the arcs to be nonnegative we could get an unbounded solution (since some circuits may have a negative cost), or a solution that was impossible to transform into a feasible solution to the path formulation (since it had a smaller value). In other words, the set of constraints *(5.7)* assures that the original space is not relaxed: only positive flows are still allowed.

With this methodology, we are adding more variables and turning a set of constraints into a set of ranged constraints. In turn, with the same set of paths in the RMP and with the circuits, we are considering implicitly a much larger number of extreme points of the subproblems, since, besides the paths that are present in the RMP, we are considering all the paths that can be obtained by adding circuits to them. We note that the same circuit, when combined with different paths, may allow the implicit consideration of several paths.

### 5.4.2 Dealing with negative cost cycles

The dual variables of constraints *(5.7)* contribute with a negative value to the modified cost of the arcs in the subproblem objective function, leading to arcs with a negative modified cost, and even to negative cost cycles. Note that this never happens in the column generation procedure for the path formulation, because it does not have any "greater than or equal to" constraints.

Formally, we define the nonnegative dual variable associated with constraint *(5.7)* of arc $ij$ as $v_{ij}$. A path $p$ of a commodity $k$ is now attractive if

$$\sum_{ij \in A} y_{ij}^{pk}(w_{ij} - v_{ij} + c_{ij}) < \pi^k.$$

The subproblem of a commodity $k$ still is a shortest path problem but now in a network that can have negative cycles (all arcs traversed in the direction of their orientation), which is NP-hard. In order to avoid solving a NP-hard problem, we can overcome this issue by solving a shortest path problem with a label correcting algorithm, and, when a negative cycle is detected, we add the associated variable to the RMP. After reoptimising the RMP, the negative cost cycle previously detected will never be generated again.

In this way subproblems may suggest paths or cycles to the RMP. This issue was previously discussed in the context of a branch-and-price algorithm for the integer MFP in Chapter 3, subsection 3.3.3 (page 70).

Note that these cycle variables are not related with the extra circuit variables that are present from the very beginning in the RMP. All the arcs of the cycles are forward arcs. Since the arc costs are nonnegative, there is an optimal solution where all the cycle variables have a null value.

### 5.4.3 Obtaining the optimal solution of the path formulation

We now present a procedure to perform the conversion of one optimal solution of the extended model to one optimal solution of the path model, with the same cost. The main idea is to redirect all the flow on circuits to paths.

(a) Select a circuit $s$ with positive flow ($d_s > 0$) and its backward arc $ij$. If there are no such circuit then stop: the actual solution is optimal to the path model.

(b) While $d_s > 0$ and there exists a path $p$ of a commodity $k$, which includes the arc $ij$, with positive flow ($\lambda^{pk} > 0$), then do the following: if $\lambda^{pk} > d_s$ then transfer $d_s$ units of flow to the path (commodity $k$) obtained by adding $p$ (commodity $k$) and $s$, let $\lambda^{pk} = \lambda^{pk} - d_s$ and $d_s = 0$; else transfer $\lambda^{pk}$ units of flow to the path (commodity $k$) obtained by adding $p$ (commodity $k$) and $s$, let $\lambda^{pk} = 0$ and $d_s = d_s - \lambda^{pk}$.

(c) If $d_s > 0$ select a circuit $t$ and an arc $ab$ such that $\gamma_{ij}^{s} = 1$, $\gamma_{ab}^{t} = -1$ and $d_t > 0$. Let $ij = ab$ and $s = t$ and go to step (b).

(d) Go to step (a).

If there are no circuit variables with positive values then the actual solution is optimal to the path model. Otherwise, the algorithm starts by selecting a circuit variable with positive flow. By definition of the circuits, one of the arcs of that circuit is a backward arc. By the constraint *(5.7)* of that arc, there must be a positive flow on paths or in other circuits. If there are positive flows on paths, we transfer the flow of the circuit to other paths. We note that the order in which the paths are considered is irrelevant, since we always get a solution that has the same value and maintains the feasibility with respect to all constraints. If the flow of the circuit is still positive after transferring the flows of the paths, then constraint *(5.7)* of arc $ij$ only has circuit variables with positive values. In that case, since circuit $s$ has a positive flow and a coefficient $-1$ in that constraint, there must be, at least, one circuit $t$ with coefficient $+1$ in that constraint, such that the constraint is not violated. Thus, the circuit $t$ of step (c) always exists. By definition of circuits, circuit $t$ has one backward arc, which is different from arc $ij$ (if it were the same its coefficient in the constraint *(5.7)* of arc $ij$ would not be $+1$).

In each outer loop (steps (a) to (d)) the flow of, at least, one circuit is reduced down to zero. Thus, at the end, we will get an optimal solution with flows exclusively on paths.

### 5.4.4 Comparison with standard column generation

There are two main additional steps in our procedure when compared to the standard column generation algorithm that was briefly described in subsection 5.3.1. The first one is that in the construction of the first RMP, we add all the circuit variables to it. The second one is that

after obtaining an optimal solution to the extended model, we need to recover a solution for the path model with flows only on paths.

Besides those algorithmic differences, the master problem and the subproblem also present some modifications, described above, that we now summarise in terms of advantages and disadvantages of our method when compared with standard column generation.

The disadvantages are: the extended RMP has ranged constraints; the subproblem can have arcs with negative costs (the algorithm used to solve it becomes less efficient) and also negative cycles (it may be necessary to insert cycles in the RMP). The main advantage is that we are implicitly considering a much larger number of extreme points in all the RMPs that we have to solve, thus we can expect to solve the problem in less iterations (that is, solve a smaller number of RMPs).

Our computational results, which will be presented later, clearly support that the advantage overcomes the disadvantages. Furthermore, they show that, in spite of the ranged constraints and of the larger number of variables (at least in the first iterations), the average time spent on solving the RMPs is smaller in the accelerated column generation algorithm when compared with the standard one. The extra variables seem to make the RMPs easier to solve. Besides that, we note that our test program uses a linear programming general solver (ILOG, 1999) to optimise the RMPs. Taking into account the level of efficiency and robustness that this kind of software has achieved in recent years (Bixby et al., 2000), the disadvantages may be regarded as more theoretical than practical.

So far, we neglected the fact that the number of circuit variables can be very large. In fact, in a general network, it is exponential with respect to the dimension of the network. A strategy to deal with that issue is to choose a subset of circuits, heuristically or by defining some preprocessing rules. An alternative is to allow deletion and insertion of circuit variables during the optimisation process.

Here we just present results for instances defined in planar graphs. In this case, we can identify a polynomial number of simple circuits such that all the graph circuits can be represented as their nonnegative combinations. Based on those circuits, we can define a polynomial number of circuit variables. Those circuit variables may be seen as elementary, in the sense that by their nonnegative combinations we can obtain other circuit variables. In fact, in our column generation algorithm for planar networks, we add a tractable number of circuit variables (the elementary ones) to the first RMP, and maintain them during all the column generation procedure, allowing the RMP to perform those nonnegative combinations, considering implicitly a much larger number of circuit variables.

In Section 5.5, we will demonstrate that the number of elementary circuit variables is polynomial.

### 5.4.5 Interpretation in the context of the Dantzig-Wolfe decomposition

The path formulation can be seen as the result of applying a DWD to the arc formulation, defining the subproblem with the flow conservation constraints. In this way, each path of a commodity is one extreme point of the subproblem of that commodity. In the master, the flow conservation constraints may be seen as convexity constraints.

In the column generation procedure, when optimising the RMP, we are selecting the best feasible (in the sense that the constraints that were kept in the master must be satisfied) solution that is obtainable by the convex combination of the extreme points of the subproblems generated so far.

In this context, the circuits that we add to the RMP can be seen as vectors vectors which, when added to some extreme points, allow us to reach feasible points that may be not reachable only with the extreme points present in the RMP. Also, those vectors can be added to extreme points of different subproblems. The set of constraints *(5.7)* is there to force the resulting point(s) to be feasible.

### 5.4.6 Example

We consider an instance of the MFP with two commodities defined over the network of Figure 5.3. The first commodity has demand *5*, origin *1* and destination *4*; the second commodity has demand *4*, origin *1* and destination *3*. Next to each pair of arcs, their cost ($c_{ij} = c_{ji}$) and their capacity ($u_{ij} = u_{ji}$) are given by this order.

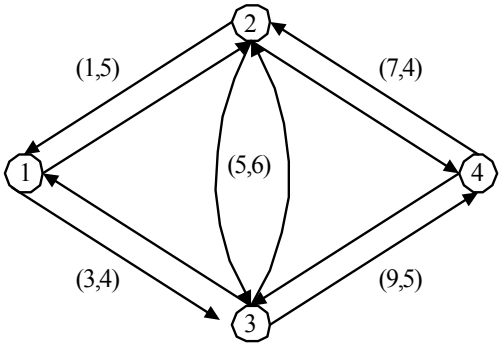



Figure 5.3 Some data of one instance of the multicommodity flow problem.

All circuits belonging to *D* are represented in Table 5.1.

| *arc/circuit* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *15* | *16* | *17* | *18* | *19* | *20* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *12* | | | −1 | 1 | 1 | | | | | | | | | 1 | 1 | 1 | −1 | | | |
| *21* | 1 | 1 | | | | −1 | | | | | | | −1 | | | | | 1 | 1 | 1 |
| *13* | 1 | | 1 | | −1 | | | | | | | | | −1 | | | 1 | | 1 | 1 |
| *31* | | −1 | | 1 | | 1 | | | | | | | 1 | | 1 | 1 | | −1 | | |
| *23* | −1 | | | | 1 | 1 | 1 | 1 | | | | −1 | | | | | | | | |
| *32* | | 1 | 1 | −1 | | | | | −1 | 1 | 1 | | | | | | | | | |
| *24* | | | | | | | −1 | | | | 1 | 1 | 1 | 1 | 1 | | | | | −1 |
| *42* | | | | | | | | 1 | 1 | −1 | | | | | | −1 | 1 | 1 | 1 | |
| *34* | | | | | | | 1 | | 1 | | −1 | | | | −1 | | 1 | 1 | | 1 |
| *43* | | | | | | | | −1 | | 1 | | 1 | 1 | 1 | | 1 | | | −1 | |

Table 5.1 Circuits of the network of the example belonging to $D$.

We form the first RMP by considering all the circuits, the paths *1-3-4* and *1-2-4* for commodity *1* and the path *1-3* for commodity *2*. The optimal solution of the RMP, which is also the optimal solution to the master problem, is $\lambda^{11}=1$, $\lambda^{21}=4$, $\lambda^{12}=4$, and $d^5=1$. This optimal solution can be converted to $\lambda^{11}=1$, $\lambda^{21}=4$, $\lambda^{12}=3$, and $\lambda^{22}=1$ (the second path of commodity *2* the path *1-2-3*) or to $\lambda^{21}=4$, $\lambda^{12}=4$, and $\lambda^{31}=1$ (the third path of commodity *1* being the path *1-2-3-4*). We note that if the circuits were not used, the RMP would be unfeasible, being necessary to add artificial variables or more paths to it and to (re)optimise it.

## 5.5 Planar Networks

Our objective in the current Section is to show how an elementary set of circuit variables can be found and that their number is polynomial, when the MFP instance is defined on a planar graph.

We give a brief, informal, introduction to some concepts related to planar graphs. A formal and deeper treatment of this subject can be found, for instance, in (Behzad et al., 1979).

Roughly speaking, a graph $G$ is said to be planar if it can be drawn on a plane without any intersection of its edges (except in the vertices). If we consider such a drawing of $G$, we define a region of $G$ as a maximal portion of the plane for which any two points can be joined by a curve that does not cross any edge or vertex. Any planar graph has an unbounded region, which is called exterior region.

Any connected planar graph with $p$ vertices and $q$ edges has $2-p+q$ regions (Euler's

formula). Also, if $p \geq 3$, then $q \leq 3p-6$. So far, we considered a drawing of the graph (more formally, a geometric embedding of the graph into the plane). It is also possible to represent a planar graph by a combinatorial embedding (the adjacency lists of the vertices are sorted according to a fixed geometric embedding) that can be obtained in polynomial time (Hopcroft and Tarjan, 1974). In the development of our method, we only need such a representation of the graph. A survey of the subject of planarising graphs can be founded in (Liebers, 2001).

In order to derive the elementary circuit variables introduced previously and to prove that, for planar networks, their number is polynomial, we consider all regions of the graph on which is based the MFP. Associated with each region there is a set of edges that form its boundary. Since our network is directed, the number of arcs $m$ is limited by $2(3n-6)$ where $n$ is the number of nodes. We associate a set of circuits with the boundary of each region. Since we are dealing with a directed graph, we replace each edge by two arcs (one in each direction). Furthermore, by definition of the circuit variables, the boundary of a region can originate more than one circuit variable. For instance, the network represented in Figure 5.4 has two regions. Considering one of them (in this case their boundary is the same), we will get six circuit variables: *21-23-31*, *12-32-31*, *12-23-13*, *12-13-32*, *21-31-32*, and *21-13-23*.

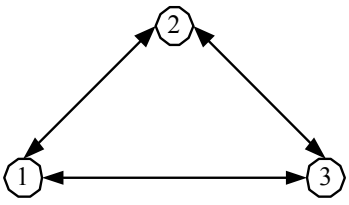



Figure 5.4 An example of generating circuits from a region.

Given that $2-n+m$, $m$, and $2m$ are upper bounds for, respectively, the number of regions that may be used in deriving circuit variables, the number of arcs of a region, and the number of different circuit variables based on the same region, an upper bound for the total number of circuit variables is

$$(2-n+m).2m.$$

Replacing $m$ with its upper bound given by $2(3n-6)$, we get

$$(5n-10).(12n-24),$$

which is a polynomial.

## 5.6 Computational Tests

We implemented the proposed method for accelerating column generation and did some computational experiments with two objectives. The main objective was to test if there is a practical improvement of the proposed method over the standard column generation procedure. The second objective was to evaluate column generation efficiency when compared with other specialised code for MFPs based on a bundle method (Frangioni and Gallo, 1999) and with a general-purpose linear programming solver, namely Cplex 6.6 (ILOG, 1999).

Our code was developed in C++ using the development environment Microsoft Visual Studio 6.0. We used the class library LEDA 4.1 (Mehlhorn and Näher, 1999) for identifying the circuits and also to solve the shortest path subproblems. We used Cplex 6.6 to solve the RMPs.

The Bundle code was provided to us by Professor Antonio Frangioni. We note that this code can be used with more general instances (namely instances with multiple origins and destinations). We also note that the adjustment of its parameters was, by no means, exhaustive and not done by an experienced user.

In spite of the theoretical closeness of the two methods (column generation and bundle), the implementations tested have a major difference: the column generation uses a disaggregated approach and the bundle an aggregated one. Both methods, taking a dual perspective of the column generation method, aim at maximising the dual Lagrangean function (obtained by relaxing the capacity constraints). That function is decomposable by commodity, thus, in each iteration, the subproblem can return (to the master) subgradients associated with the commodities (disaggregated version) or return one subgradient associated with the whole function (aggregated version). In the column generation algorithm implemented, the subproblem returns one extreme point per commodity (or dually, one subgradient associated with the component of one commodity of the decomposable dual Lagrangean function). In the case of the bundle implementation used in these tests, the subproblem returns one subgradient of the whole dual Lagrangean function.

It is worth noting that a disaggregated bundle algorithm could also be implemented, which might significantly improve its solution times.

We refer to the different programs described above as: CG (column generation), CGA (column generation with acceleration), Cplex (the problem, formulated with flows on arcs, by using only Cplex 6.6) and Bundle.

All the reported results were obtained on a personal computer equipped with a Pentium 4, 2 GHz processor, and 1 GB of RAM, running Windows XP Professional Edition. All the time values presented are expressed in seconds.

### 5.6.1 Test instances

In order to test the empirical efficiency of the developed methodology, we performed some computational tests on three groups of instances. Two of them, which we refer to as *A* and *B*, were generated with a random generator for MFPs in planar graphs, implemented in C++ using the development environment Microsoft Visual Studio 6.0 and the class library LEDA 4.1 (Mehlhorn and Näher, 1999). The other one comes from (Larsson and Yuan, 2004) and was obtained in (Frangioni, 2005). We refer to this last set of instances as the *Planar* instances, in order to keep their original designation, while noting that all the instances in all sets are planar.

For the two first groups of instances (*A* and *B*), the underlying graph is constructed by using the LEDA functions *random_planar_map* and *random_planar_graph*, which we now briefly describe. First, *n* random points in the unit square are generated and their triangulation is computed. Then *m* arcs are kept. Instances *A* are generated considering undirected arcs that are replaced by two directed arcs between the same pair of nodes. Thus, for this group of instances, if an arc *ij* exists then the arc *ji* also exists and both have the same cost and capacity. Instances *B* are generated in a similar way. The difference is that the undirected arcs are replaced with just one directed arc, thus if an arc *ij* exists then the arc *ji* does not exist. For both groups of instances the cost of each arc is the Euclidean distance between its origin and destination nodes. Each commodity is defined by its origin-destination pair, in a random way (without repetitions). Their demands are randomly chosen up to a maximum value given as input. In order to generate only feasible instances, some capacities are calculated as the total flow of arcs given by the shortest path solution of each commodity. The others are randomly assigned with values between two input parameters.

For each group, we generated two sets of instances, one with 100 nodes and the other with 600 nodes. For each of these sets we generated four subsets with different densities and different numbers of commodities. For each subset, we generated three instances, for which we present average results. The number of arcs and commodities for each subset are given in Table 5.2 and Table 5.3. The name of the instance has the following meaning: group, number of nodes, density (*s* − sparse, *d* − dense) and a qualitative measure of the number of commodities (*s* − small, *l* − large). A dense graph has a number of arcs close to its maximum in a planar graph, that is, *2(3n−6)* for instances of group *A* and *3n−6* for instances of group *B*.

The number of nodes, arcs, and commodities of the planar instances are given in Table 5.4.

| *Instances* | *A100dl* | *A100ds* | *A100sl* | *A100ss* | *A600dl* | *A600ds* | *A600sl* | *A600ss* |
|---|---|---|---|---|---|---|---|---|
| *m* | 569 | 572 | 400 | 400 | 3562 | 3557 | 2400 | 2400 |
| *h* | 600 | 300 | 600 | 300 | 3600 | 1800 | 3600 | 1800 |

Table 5.2 Average number of arcs and number of commodities of group *A* instances.

| *Instances* | *B100dl* | *B100ds* | *B100sl* | *B100ss* | *B600dl* | *B600ds* | *B600sl* | *B600ss* |
|---|---|---|---|---|---|---|---|---|
| *m* | 285 | 286 | 200 | 200 | 1771 | 1778 | 1200 | 1200 |
| *h* | 600 | 300 | 600 | 300 | 3600 | 1800 | 3600 | 1800 |

Table 5.3 Average number of arcs and number of commodities of group *B* instances.

| *Instance* | *planar30* | *planar50* | *planar80* | *planar100* | *planar150* |
|---|---|---|---|---|---|
| *n* | 30 | 50 | 80 | 100 | 150 |
| *m* | 150 | 250 | 440 | 532 | 850 |
| *h* | 92 | 267 | 543 | 1085 | 2239 |

Table 5.4 Number of nodes, arcs and commodities of the planar instances.

### 5.6.2 Preliminary tests

The general-purpose linear programming solver Cplex 6.6 can use different algorithms to optimise a linear programming problem, namely simplex primal (referred to in this text as P) and dual (D), hybrid primal (HP) and hybrid dual (HD), and a barrier algorithm (B). Hybrid algorithms use an advanced basis obtained by solving the network type part of the problem and then simplex primal or dual. We ran preliminary tests to determine which of these alternatives results in a smaller computational time in obtaining the optimal solution for our code (that uses Cplex 6.6 callable library in order to solve the RMPs) and for solving the problem, formulated with flows on arcs, by only using Cplex 6.6.

We now present the results from the preliminary tests for each of the methods.

**Cplex**

We tested the five algorithms with a few instances in order to select one algorithm to use in the tests of all instances. We kept all the parameters of Cplex in their default values. In particular, the Cplex optimality tolerance was not changed, thus being equal to *1e–6*.

The instances that we tested were: *A100dl1*, *B100dl1* (larger instances with 100 nodes),

*A100ss1*, *B100ss1* (smaller instances with 100 nodes) and the *planar* instances. For instances with 600 nodes, Cplex could not solve the problem due to the huge size of the models. The same happened with the larger planar instance (*planar150*). For almost all the other planar instances, the dual algorithm resulted in the best computational times. The exception was the *planar50* instance, for which the best result was obtained when using the primal algorithm.

The results for the tested instances with 100 nodes are given in Table 5.5.

| *Instance* | *Algorithm* | | | | |
|---|---|---|---|---|---|
| | *P* | *D* | *HP* | *HD* | *B* |
| *A100dl1* | 4467.3 | 188.5 | 2479.9 | 481.4 | 267.3 |
| *A100ss1* | 294.1 | 24.4 | 181.4 | 54.0 | 91.3 |
| *B100dl1* | 87.9 | 250.8 | 83.3 | 149.7 | 17.1 |
| *B100ss1* | 1.8 | 1.5 | 1.3 | 1.5 | 1.9 |

Table 5.5 Cplex results of the preliminary tests.

After those tests, we decided to test each subset of instances with the best algorithm, since the preliminary tests with one of the smaller and one of the larger instances indicated different algorithms. Thus, we carried out also a preliminary test with one instance from the subset *B100ds* and one other from *B100sl*. For the instance from the subset *B100ds* Barrier was the best algorithm (5.3 seconds against the 16.1 of the second best that was Hybrid Primal) and for the instance from the subset *B100sl* Hybrid Primal performed better than all the others (3.0 seconds against 3.4 of the second best that was the Primal).

**Bundle**

We calibrated a few parameters of the Bundle in one instance of each subset of *A100* and *B100*. We set at *1e−5* the relative tolerance desired and we accepted (dual) solutions for which the final norm of the direction was smaller than *1e−3* (controlled through the $t^*$ parameter – for a description of the parameters of the used implementation of the bundle method, we refer the interested reader to the reference (Crainic et al., 2001)). We tried, with a few runs, to adjust the initial $t$, choose between $m_1=0.1$ and $m_1=0$ and also select the most efficient shortest path algorithm for the solution of the subproblems. The other parameters of the algorithm were set to their default values (chosen by the developer of the code).

Being a dual method, the obtainable primal solution may be slightly unfeasible. With the parameter setting just described, that solution is *1e−5*-optimal and, the violation of the constraints is, at most, *1e−3*.

We did not perform tests with the Bundle for instances with 600 nodes, given that for one

of the smaller instances with that number of nodes (*B600ss1*) the algorithm reached the maximum number of iterations set by us (*10000*), taking more than 1500 seconds.

We did not test the *planar* instances, since previous computational results with standard column generation and bundle methods were reported in (Larsson and Yuan, 2004).

**Standard column generation**

In both column generation algorithms (standard and accelerated), we used the same stopping criterion: the algorithm stops when there are no attractive columns within a tolerance of *1e-5*. This means that, in the worst case, the solution obtained is *h.(1-e5)*-optimal (we remember that *h* is the number of commodities/subproblems), since the sum of the reduced costs given by the most attractive columns of each of the subproblems is precisely a duality gap (as can be proved through the equivalence of Lagrangean relaxation and DWD).

In Table 5.6 we present the results of the preliminary tests for standard column generation. For the *A100*, *B100*, and *B600* instances, we decided to run the primal algorithm. For the *A600* instances, we decided to run both primal and dual on the four subsets of instances and to choose the best result for each. For the instance *B600dl1*, the number of columns needed to obtain an optimal solution was significantly larger than for the other instances, so we also tested (for that instance, with the best algorithm, that is the primal) the removal of columns with a positive reduced cost in every iteration. That test is signalled with '*' in Table 5.6. Its result shows that the alternative in question is not efficient. The primal algorithm resulted in the best computational time for all the planar instances.

**Accelerated column generation**

In Table 5.7 we present the results of the preliminary tests for CGA. For all the instances, the primal algorithm was the most efficient. Again, for the instance *B600dl1*, the number of columns needed to obtain an optimal solution was significantly larger than for the other instances, so we also tested the removal of columns with a positive reduced cost in every iteration. That test is signalled with '*' in Table 5.7. Its result shows that performing the removal of columns (all that have positive reduced cost except the ones corresponding to circuits) is clearly preferable. The primal algorithm resulted in the best computational time for all the planar instances.

| *Instance* | *Algorithm* | | | | |
|---|---|---|---|---|---|
| | *P* | *D* | *HP* | *HD* | *B* |
| *A100dl1* | 1.5 | 1.6 | 6.6 | 2.2 | 9.9 |
| *A100ss1* | 0.5 | 0.6 | 1.9 | 0.7 | 3.0 |
| *A600dl1* | 168.8 | 188.5 | 2252.4 | 339.3 | >2000 |
| *A600ss1* | 258.0 | 174.4 | >2000 | 339.2 | 1573.5 |
| *B100dl1* | 2.3 | 5.3 | 7.4 | 7.1 | 9.0 |
| *B100ss1* | 0.2 | 0.3 | 0.7 | 0.4 | 1.6 |
| *B600dl1* | 1326.6 | 3090.7 | 11936.7 | 1837.6 | 1326.6 |
| *B600dl1* | 9523.5* | | | | |
| *B600ss1* | 74.3 | 120.2 | 634.4 | 602.1 | 341.3 |

Table 5.6 Standard column generation results of the preliminary tests.
* Removal of columns with positive reduced cost in every iteration.

| *Instance* | *Algorithm* | | | | |
|---|---|---|---|---|---|
| | *P* | *D* | *HP* | *HD* | *B* |
| *A100dl1* | 1.5 | 1.5 | 10.7 | 2.4 | 10.6 |
| *A100ss1* | 0.5 | 0.5 | 4.0 | 0.8 | 5.4 |
| *A600dl1* | 59.0 | 105.7 | 1003.0 | 122.2 | >2000 |
| *A600ss1* | 103.1 | 137.5 | 1452.8 | 196.4 | 1131.9 |
| *B100dl1* | 1.3 | 1.6 | 7.5 | 4.9 | 4.9 |
| *B100ss1* | 0.2 | 0.2 | 0.6 | 0.3 | 0.9 |
| *B600dl1* | 252.1 | 461.7 | 1962.4 | 719.4 | 252.1 |
| *B600dl1* | 161.8* | | | | |
| *B600ss1* | 24.4 | 31.2 | 222.7 | 64.5 | 95.4 |

Table 5.7 Accelerated column generation results of the preliminary tests.
* Removal of columns with positive reduced cost in every iteration.

### 5.6.3 Computational results

We now present the computational results for all instances. The results reported are the average of each subset of three instances, except for *planar* instances.

**Instances A**

In Table 5.8 and Table 5.9 we report the results for instances A. The last column in Table 5.8 is the relative variation (in percentage) of CGA with respect to CG (their values are

calculated with more precision than the absolute times presented).

| *Instance* | *Cplex* | *Bundle* | *CG* | *CGA* | *%Δ* |
|---|---|---|---|---|---|
| *A100dl* | 195.2 | 18.2 | 1.2 | 1.1 | –10.2 |
| *A100ds* | 29.7 | 9.4 | 0.6 | 0.7 | 31.9 |
| *A100sl* | 77.1 | 5 | 0.8 | 0.8 | –4.1 |
| *A100ss* | 23.8 | 2.9 | 0.5 | 0.5 | 5.3 |
| *A600dl* | – | – | 154.4 | 52.8 | –65.8 |
| *A600ds* | – | – | 82.7 | 30.9 | –62.6 |
| *A600sl* | – | – | 173.5 | 98.4 | –43.3 |
| *A600ss* | – | – | 54.9 | 34.4 | –37.4 |

Table 5.8 Time results for *A* instances.
– Not tested.

| *Instance* | *Number of iterations* | | | *Number of columns* | | *Mean time RMPs* | |
|---|---|---|---|---|---|---|---|
| | *Bundle* | *CG* | *CGA* | *CG* | *CGA* | *CG* | *CGA* |
| *A100dl* | 601* | 8.0 | 7.3 | 2228 | 2782 | 0.08 | 0.09 |
| *A100ds* | 422* | 8.7 | 7.7 | 1155 | 2087 | 0.02 | 0.06 |
| *A100sl* | 428* | 8.0 | 7.7 | 2046 | 2831 | 0.05 | 0.05 |
| *A100ss* | 239* | 8.7 | 7.7 | 1143 | 1815 | 0.03 | 0.04 |
| *A600dl* | – | 12.3 | 8.7 | 18551 | 19262 | 10.0 | 3.7 |
| *A600ds* | – | 11.0 | 8.7 | 10867 | 13761 | 6.0 | 2.2 |
| *A600sl* | – | 11.3 | 10.3 | 18863 | 20593 | 13.4 | 7.9 |
| *A600ss* | – | 11.0 | 11.0 | 9295 | 14180 | 3.9 | 2.2 |

Table 5.9 Detailed results for *A* instances.
Values marked with * are for the first instance of the corresponding set.
– Not tested.

**Instances B**

In Table 5.10 and Table 5.11 we report the results for the *B* instances.

| *Instance* | *Cplex* | *Bundle* | *CG* | *CGA* | *%Δ* |
|---|---|---|---|---|---|
| *B100dl* | 20.1 | 38.3 | 2.1 | 1.2 | –44.9 |
| *B100ds* | 7.2 | 5.3 | 0.7 | 0.5 | –31.1 |
| *B100sl* | 3.5 | 10.6 | 0.6 | 0.4 | –38.2 |
| *B100ss* | 1.5 | 2.2 | 0.3 | 0.2 | –1.3 |
| *B600dl* | – | – | 1470.9 | 175.1 | –88.1 |
| *B600ds* | – | – | 685.3 | 166 | –75.8 |
| *B600sl* | – | – | 255.1 | 78.9 | –69.1 |
| *B600ss* | – | – | 75.9 | 26.9 | –64.6 |

Table 5.10 Time results for *B* instances.
– Not tested.

| *Instance* | *Number of iterations* | | | *Number of columns* | | *Mean time RMPs* | |
|---|---|---|---|---|---|---|---|
| | *Bundle* | *CG* | *CGA* | *CG* | *CGA* | *CG* | *CGA* |
| *B100dl** | 1462* | 8.3 | 5.7 | 3272 | 2257 | 0.22 | 0.13 |
| *B100ds** | 295* | 10.0 | 7.0 | 1779 | 1394 | 0.05 | 0.04 |
| *B100sl** | 962* | 6.7 | 5.3 | 2593 | 1612 | 0.06 | 0.03 |
| *B100ss** | 340* | 6.3 | 6.0 | 1173 | 868 | 0.02 | 0.01 |
| *B600dl* | – | 13.3 | 48.3** | 39636 | 8880** | 109.1 | 3.1 |
| *B600ds* | – | 16.3 | 10.3 | 24791 | 15357 | 41.3 | 15.3 |
| *B600sl* | – | 11.3 | 6.3 | 27958 | 17465 | 21.5 | 10.4 |
| *B600ss* | – | 12.0 | 7.3 | 13057 | 8673 | 5.5 | 2.7 |

Table 5.11 Detailed results for the *B* instances.
* First instance of the corresponding set.
** We recall that, in all iterations, removal of columns is performed.
– Not tested.

**Planar instances**

In Table 5.12 and Table 5.13 we report the results for the planar instances.

| *Instance* | *Cplex* | *CG* | *CGA* | *%Δ* |
|---|---|---|---|---|
| *planar30* | 0.2 | 0.1 | 0.1 | –14.3 |
| *planar50* | 3.9 | 0.2 | 0.2 | 11.8 |
| *planar80* | 73.0 | 1.3 | 0.8 | –38.4 |
| *planar100* | 291.2 | 3.1 | 2.0 | –35.5 |
| *planar150* | – | 88.6 | 43.6 | –50.7 |

Table 5.12 Time results for the planar instances.

| *Instance* | *Number of iterations* | | *Number of columns* | | *Mean time RMPs* | |
|---|---|---|---|---|---|---|
| | *CG* | *CGA* | *CG* | *CGA* | *CG* | *CGA* |
| *planar30* | 4 | 4 | 223 | 500 | 0.01 | 0.01 |
| *planar50* | 9 | 8 | 806 | 1290 | 0.00 | 0.01 |
| *planar80* | 9 | 7 | 2410 | 2632 | 0.10 | 0.07 |
| *planar100* | 9 | 7 | 4256 | 4372 | 0.23 | 0.18 |
| *planar150* | 14 | 11 | 14900 | 12746 | 6.0 | 3.7 |

Table 5.13 Detailed results for the planar instances.

We now summarise the conclusions that can be drawn from the computational tests just presented.

When compared with Cplex, standard column generation is a very efficient method to solve the tested type of instances. In the smaller instances, it was always faster (at least five times but frequently much more). For the larger instances, which Cplex could not solve with the available memory, standard column generation could do it in reasonable amounts of time.

When compared with the bundle method, standard column generation is also more efficient for the instances tested. We note that the fact that the bundle code used is an aggregated implementation of the method may be a significant reason for its poor results, as the number of iterations is extremely large when compared with column generation.

The computational tests clearly show the effectiveness of the method proposed in this work to accelerate column generation. In fact, for all groups of instances that took more than two seconds to be solved by standard column generation, the relative improvement is always greater than 35% and frequently greater than 60%.

The best relative improvements were achieved in the larger instances, and, in particular, in the instances with a large number of commodities defined in dense networks.

We expected a smaller number of iterations (that is the number of RMPs solved) for the proposed method, when compared to standard column generation. For almost all sets of

instances, that was the case, but the difference was not as significant as expected. As for the number of columns, the results show that the insertion of extra variables does not change significantly the size of the RMPs; in fact, sometimes the final number of columns is inferior in the accelerated column generation case. We expected the time spent in solving the RMPs to increase in the case of the extended model, due to the presence of extra constraints (in fact, ranged constraints). However, that did not happen: the presence of the extra variables made the solution of the RMPs easier, as can be seen in the average time spent on the RMPs.

## 5.7 Conclusions

In this Chapter, we presented a way of accelerating column generation for the linear MFP in planar graphs. The method used is based on a new model, which includes a polynomial number of extra variables corresponding to flows on circuits. After an optimal solution to the model with extra variables is obtained, we recover an optimal solution to the original model by a procedure that forces all the extra variables to have a null value by redirecting its flow to the original variables, which are flows on paths.

Computational tests were made for three sets of randomly generated instances: two of them, generated by us, with 24 instances (several groups of three instances with similar characteristics) and the third, not generated by us, with five instances. The results of these tests empirically showed that our method to accelerate column generation is effective. In almost all instances, our procedure reduced the computational time by significant amounts. In particular, for all groups of instances that took more than two seconds to be solved by standard column generation, the relative improvement was always greater than 35% and frequently greater than 60%.

The presented approach poses no theoretical difficulties when applied to other multicommodity flow problems. In particular, the extension to instances with multiple origins and destinations is trivial.

A natural development of the current work is to apply the same approach on multicommodity problems defined in general networks. The main question that arises is how to control the (exponential) number of extra variables, or, in other words, how to select an effective subset of extra variables such that they do not render the master problem too large.

# 6 *ADDing*: Automatic Dantzig-Wolfe Decomposition for Integer Column Generation

In this Chapter, we describe *ADDing*, an implementation of a general branch-and-price algorithm in C++.

The main distinctive feature of *ADDing* is that it can be used as a "black-box": all the user is required to do is to provide an original (mixed) integer model. *ADDing* automatically decomposes the original model and combines column generation and branch-and-bound (branch-and-price) to obtain an (integer) optimal solution. All the (non-trivial) implementation details of such type of algorithms (such as interaction between the restricted master problem and the subproblem(s), combination of column generation and branch-and-bound, rows and columns management, management of the search tree, data structures, ...) are transparent to the user, although controllable by a set of input parameters.

*ADDing* can also be customised to meet a specific problem, if the user is willing to provide a subproblem solver and/or specific branching rules. Those can be implemented with a few functions.

## 6.1 Introduction

*ADDing*, *Automatic Dantzig-Wolfe Decomposition for INteger column Generation*, is an implementation of a general branch-and-price algorithm in C++. The main objective of its development was to provide an easy and fast way to implement (different) decomposition approaches for (mixed) integer problems.

Branch-and-price combines two well-established methods, column generation and branch-and-bound, to obtain the optimal solution of (mixed) integer problems. Although those two methods are known since the late 1950s, only in the middle 1980s their first combination was developed to obtain optimal integer solutions for a routing problem (Desrosiers et al., 1984) and only in the late 1990s the first revision paper about branch-and-price was published (Barnhart et al., 1998). Branch-and-price methods were reviewed in Chapter 2; other surveys, besides the one already mentioned, can be found in (Wilhelm, 2001; Lübbecke and Desrosiers, 2002).

In its more simple use, the main feature of *ADDing*, is that its user only needs to specify an original formulation for the (mixed) integer problem (MIP) he/she wants to solve, along with the decomposition to be used (that is, which constraints define the subproblem(s)). All the (non-trivial) implementation details of such type of algorithms (such as interaction between the restricted master problem (RMP) and the subproblem(s), combination of column generation and branch-and-bound, rows and columns management, management of the search tree, data structures, ...) are hidden and the user does not need to worry about them. Being so, *ADDing* can be used as a "black-box" where the input is a MIP model and the specification of a decomposition and the output is an optimal solution obtained by branch-and-price (of course, if the problem has one, and a time limit or other stopping conditions specified by the user through parameters were not met).

*ADDing* can also be customised for a specific problem by letting its user implement two major pieces of branch-and-price algorithms: the subproblem solver and branching rules. This allows the full exploration of the structure of the problem at hand. In this type of use, one exact subproblem solver must be provided. An unlimited number of subproblem heuristics may also be provided; these may be used in constructing the first RMP and in solving the subproblems heuristically. Specific branching rules play a fundamental role in an efficient search of the tree, and can also be provided by the user. Being so, besides the "black-box" use, *ADDing* can also

be used as a "tool-box", where the user only implements small blocks of code related with the specific problem he/she wants to solve.

The two types of use described in the two previous paragraphs are, by no means, exclusive. In a first approach to a structured MIP, *ADDing* can be used as a "black-box" in order to test different decompositions and, roughly, the efficiency of the branch-and-price method. That is done with a very small coding effort. The use of *ADDing* as a "tool-box" afterwards allows the improvement of the algorithm's efficiency.

The generality of *ADDing* relies on the exchange of information between the original formulation (provided by the user) and the master model (derived internally). The branching scheme consists in deriving branching constraints in the original variables and including them in the master problem, modifying the objective function of the subproblem(s) accordingly (as opposed to branching by subproblem and master modifications, as is common in branch-and-price algorithms for binary problems).

The current version of *ADDing* is *1.0*. It has been used as a "tool-box" for two different decompositions for the binary multicommodity flow problem (see Chapter 4) and, as a "black-box", for two different decompositions for a MIP (multi-item lot sizing with setup times) (Pimentel et al., 2004).

Several frameworks have been developed to make the implementation of branch-and-price algorithms (in fact, branch-and-cut-and-price algorithms) easier, such as Abacus (Thienel, 1995; Jünger and Thienel, 2000), COIN/BCP (Ralphs and Ladányi, 2001) and Symphony (Ralphs and Ladányi, 2003). A distinctive feature of *ADDing* is the "black-box" use. At least in a preliminary phase, for example when testing different decomposition approaches, it may serve as a guide to the subsequent full exploration of a branch-and-price algorithm.

The current version of *ADDing* by no means attains the power and flexibility of those frameworks (for example, it does not include the use of cuts), neither implements several components that certainly would improve branch-and-price efficiency (such as preprocessing and reduced cost fixing), some of which are detailed in (Vanderbeck, 2005). However, we believe its simplicity of use and future improvements related with issues not implemented in those frameworks (such as stabilisation techniques and multiple Dantzig-Wolfe decomposition, addressed in Chapter 2, subsection 2.5.5, page 50) justify its further development.

This Chapter is written with two purposes: to introduce to the use of *ADDing* and to describe briefly its internal structure. We do not detail the theoretical foundations behind it, which were presented in Chapter 2. Section 6.2 (along with the Appendix) can be taken as a brief users' manual for using *ADDing* as a "black-box" and as a "tool-box". In Section 6.3, the general structure and the main classes of *ADDing* are described. In Section 6.4, conclusions

from this work are drawn and future developments are discussed.

## 6.2 Using *ADDing*

In this Section we provide an overview of *ADDing* from a user perspective.

*ADDing* was programmed in *C++* in the development environment Microsoft Visual Studio 6.0 and uses Cplex (ILOG, 2002) for solving the RMPs and the subproblem(s) (when the user does not provide an exact subproblem solver).

The main purpose of *ADDing* is to provide an automatic way of solving a MIP problem by decomposing it (using Dantzig-Wolfe decomposition) and combining column generation and branch-and-bound (branch-and-price).

### 6.2.1 Models representation

It is assumed that the user has one model, denoted as the original model, which can be schematically depicted as in Figure 6.1. We omit the size of vectors and matrices for simplicity of notation. Rows SP refer to the constraints that define the subproblems.

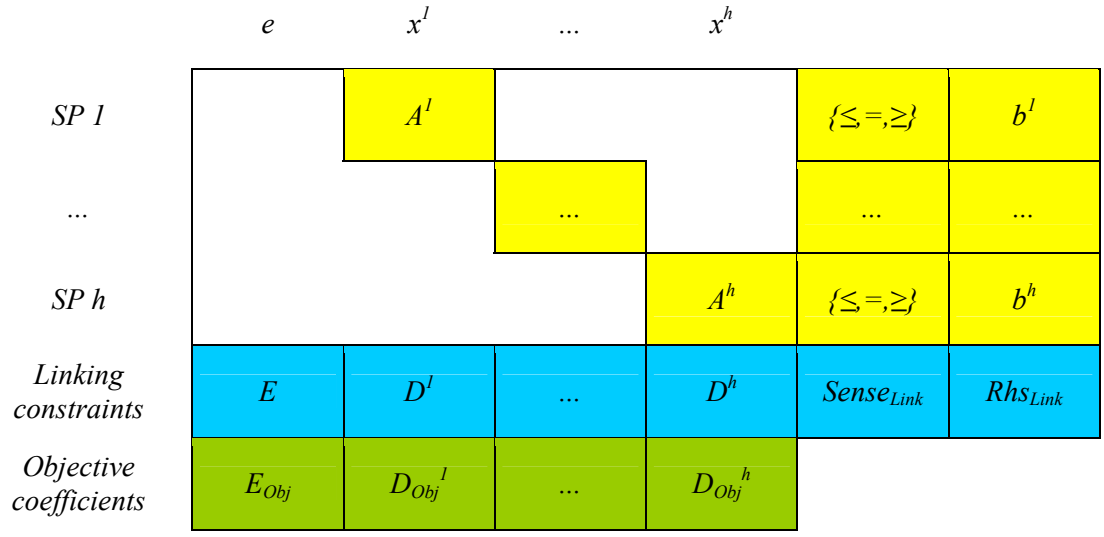



Figure 6.1 Schematic representation of an original model for *ADDing*.

We now point out some issues related with the original model and its schematic representation.

- The original model has a block angular with linking constraints structure. However, considering only one ($A$) block, that is $h=1$, a general MIP may be considered. If the only block has a block diagonal structure, we obtain an aggregated decomposition.
- A different ordering of the variables gives another representation that leads to a different decomposition.

- The *x* variables can be linear and/or integer and/or binary. In the current version of *ADDing* the *e* variables, denoted as extra variables, must be linear.
- The *e* variables, may be used as stabilisation variables (a model that uses this type of variables was presented in Chapter 5).

Based on the original model, *ADDing* builds an internal master problem, as depicted in Figure 6.2 (in fact, it does not use that representation explicitly, since, of course, columns are dynamically generated – and, depending on some input parameters, removed; the same happens with rows – however, the user does not have to worry about these internal details).

Columns *Art* and *e* are associated with artificial and extra variables, respectively. The *e* variables were already defined in the original model, and since they do not belong to any subproblem, their translation to the master problem is straightforward. As for the artificial variables, we decided to let the user specify their number and coefficients in the master model, possibly taking advantage of his knowledge of the problem. However, *ADDing* has a hidden artificial variable, allowing the user not to include artificial variables explicitly in the master model. Again the user is allowed to incorporate his/her knowledge in the master model by defining the sense of convexity constraints (for example, when the feasible region of the subproblem includes the origin, the user may prefer to define "less than or equal to" convexity constraints, as opposed to the usual equalities). Letting the user specify the right-hand side (RHS) of convexity constraints extends the possible use of *ADDing*. For example, in the path decomposition for the integer multicommodity flow problem (addressed in Chapter 3) those constraints have a RHS different from one.

In Figure 6.2 two columns ($\lambda^{k1}$ and $\mu^{k1}$) related with one extreme point ($y^{k1}$) and one extreme ray ($u^{k1}$) of a subproblem $k$ ($1 \leq k \leq h$) are represented. We note that, as shown in the same Figure, all the coefficients of these columns can be computed given the original model. We also note that if the subproblem(s) do(es) not have extreme points, but only extreme rays, there are no convexity constraints in the master model.

Summing up, the user of *ADDing* must provide all the decomposition information, which includes the one depicted with colours in Figure 6.1 and Figure 6.2 (including the dimensions of the matrices and vectors). Along with that information, the user may specify which constraints are present in the first RMP. That issue is related with the dynamic management of rows, which is controlled through a set of parameters. In some models, the *D* matrices are all equal, that is $D=D^1=...=D^h$. For those models, the user may define only one *D* matrix.

In a "black-box" use, the subproblems (represented in yellow in Figure 6.1) only need to be specified, that is, the user must provide no solver. In a "tool-box" use, the subproblems are

totally managed by the user: he/she must implement some functions in order to return solutions to them.

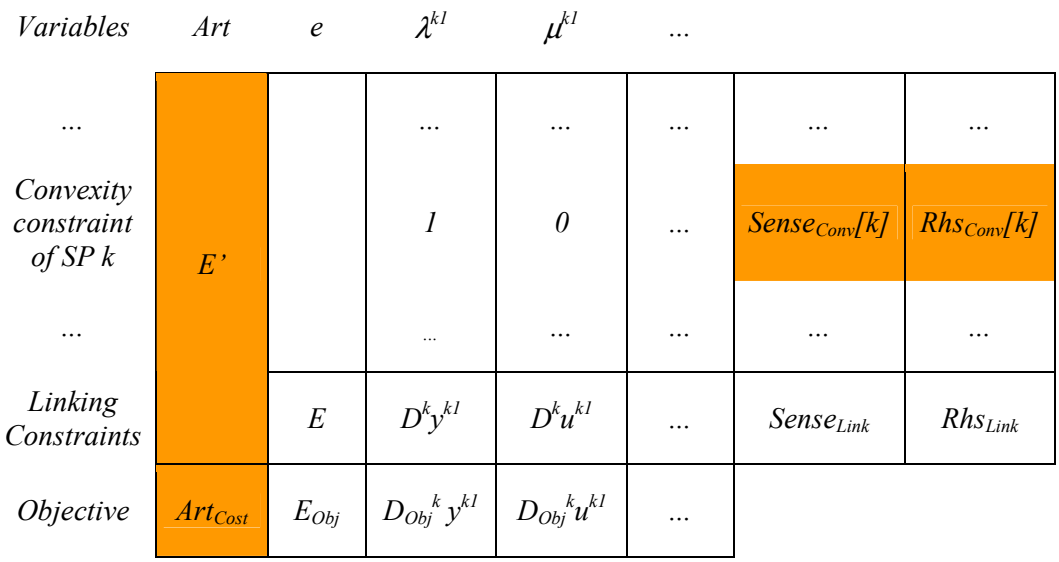



Figure 6.2 Schematic representation of the master model.

### 6.2.2 "Black-box" use

In Figure 6.3, we illustrate the use of *ADDing*. In a "black-box" use, all the user must do is to:

- Create one object of the class *Decomposition Model*.
- Use the public member functions of *Decomposition Model* to specify the decomposition (these are detailed in the Appendix, page A2).
- Create an object of the class *Branch and Price*, whose constructor has three arguments: one decomposition model object, one string with the name of the file with the parameters (detailed in the Appendix, page A5) and one string with the name of the file where the results will be written (detailed in the Appendix, page A13).
- Call the *Optimise(...)* public member function of *Branch and Price* that returns the final status of the optimisation (the possible values are given in Figure 6.4).

The public member functions of the *Decomposition Model* class allow the specification of the decomposition model to be used. Two examples of such functions, *SetDimensions(...)* and *SetTypeVars(...)*, are given in Figure 6.3. All the member functions the user must be aware of are detailed in the Appendix, page A2. Almost all of them have to do with the definition of the matrices and vectors represented with colours in Figure 6.1 and in Figure 6.2. All the user must do is to represent them in a compressed (column oriented) format (as the one used, for example, in COIN and Cplex). An issue worth pointing out is that the user may decide which rows will be

generated only if needed (that is, if they are violated in an optimal solution). This type of dynamic management of rows, which can be further controlled by some parameters, can significantly improve the solution time of the algorithm.

```
// Decomposition model
decmodel *DM=new decmodel;

// Construct the decomposition model
DM->SetDimensions(10,20,30,30);
DM->SetTypeVars(0,10,0);
// ...

// Branch-and-price solver
branchprice *BP=new
          branchprice(DM,"parameters_file.txt","results_file.txt");

int status = BP->Optimise();
cout << endl << status << endl;

delete DM;
delete BP;
```

Figure 6.3 Using *ADDing* as a "black-box": code required to the user (not including the specification of the decomposition model).

```
0: An optimal solution was found.
1: Time limit achieved in the root node.
2: Time limit achieved not in the root node.
3: The root node is unfeasible.
4: There are no integer feasible solutions.
5: A solution could not be obtained (numerical difficulties in
solving the RMP).
6: Problem is unbounded.
7: Maximum number of optimised nodes achieved.
```

Figure 6.4 Return values for *branchprice->optimise()*.

The use of user-defined parameters allows controlling several features of the branch-and-price algorithm. Their use and meaning are detailed in the Appendix, page A5. Here we point out some of the main features the user may control using parameters (additional features, related with the user customisation of *ADDing* are listed in the next subsection):

- The use of a heuristic at the root node. The heuristic consists in solving with Cplex the MIP associated with the ("optimal") RMP obtained in the root.
- The inexact solution of the RMP in some iterations.
- The dynamic management of columns and rows.
- The search strategy of the tree (depth, breadth, best, depth until an incumbent is found and then best, depth when branching occurs, best in the other situations).
- The branching variable (fractional variable with fractional part closest to 0.5, first

fractional variable found, fractional variable with fractional part closest to 1, fractional variable with fractional part closest to 0). When customising *ADDing* (see next subsection) the user may specify other branching types, namely by generating branching constraints with more than one variable (branching on hyperplanes).

- The specification of tolerances.

The output of *ADDing* is a file with the results (name given as the third argument to *branchprice::optimise()* and extension "rst") and another file (same name, extension "sol"). The results file has various information on the optimisation process, including the times spent in different parts of the algorithm, the value of the solutions, the number of optimised nodes in the tree and the number and largest dimensions of the RMPs solved. An example of such a file is given in the Appendix, page A13).

### 6.2.3 "Tool-box" use

When using *ADDing* as a "tool-box", it is up to the user to provide a subproblem solver and/or one branching scheme. For that purpose, a user class must be derived from the base class *Subproblem* (which does not have pure virtual member functions in order to allow the "black-box" use described before) (re)defining a set of virtual member functions. We denote that derived class by *MySubproblem*. We now briefly describe the fundamental issues related with its implementation; details are given in the Appendix, page A14.

When implementing a subproblem solver, there are two functions that have to be necessarily redefined by *MySubproblem*: *SetSP(...)* and *Optimise(...)*. The first one is conceived to receive the modified costs of the current iteration of the column generation algorithm. Those will be used in *Optimise(...)*, which must implement an exact subproblem solver, returning an optimal extreme point or a ray.

Other member functions of *Subproblem* may be redefined by *MySubproblem* (in fact, *defined*, since they are virtual dummy functions, the base class does not have a default implementation). Those are related with subproblem heuristics, second best solutions, and explicitly defined sets of feasible solutions of the subproblem.

Subproblem heuristics can be used for three purposes: constructing the first RMP, solving the subproblem heuristically (which can be useful if the subproblem is a "difficult" one – in that case, when the heuristics do not generate attractive columns, the exact solver is used to ensure optimality) or inserting additional attractive columns in some iterations of the column generation algorithm.

Two types of heuristics can be implemented: aggregated and disaggregated. In an aggregated heuristic the solution of one subproblem influences the solution of the others. In a

disaggregated heuristic each subproblem is solved independently. We note that the extreme points will always be inserted in the RMP in a disaggregated manner. Currently *ADDing* does not support aggregated columns (that is, the possibility of one column being related with more than one subproblem).

The *MySubproblem* class may also implement a function (*OptimiseNext(...)*) to provide the second best, third best, and so on, solutions. This function is useful for generating additional columns of good quality in some iterations of the column generation algorithm.

The first RMP may have a set of points, or rays, explicitly given by the *MySubproblem* class, which is achieved by implementing the function *GetSetExtreme(...)*.

When implementing specific branching rules, the only function that has necessarily to be redefined by *MySubproblem* is *GetBranches(...)*. The *MySubproblem* class can access the current (fractional) solution values (by using a public member function of the class *Original Solution*). Based on that solution it must construct a set of branching constraints (one for each new node), based on the original variables, through the use of the public member functions of class *Constraint* (an array of objects of the class *Constraint* are passed by reference in this function).

A set of parameters controls the use of all the features described in this subsection. All the details about the functions mentioned here and those parameters are given in the Appendix.

## *6.3* Inside *ADDing*

### 6.3.1 Overview

The main two pre-requisites we considered for the development of *ADDing* were: (i) it should be simple to use, either as a "black-box" or as a "tool-box" and (ii) its design should allow an easy incorporation of new features, compromising as little as possible its overall structure.

The concepts of inheritance and polyphormism of object-oriented programming clearly seemed to allow the accomplishment of the first pre-requisite. In addition, the modular approach of object-oriented modelling seemed to make easier the accomplishment of the second pre-requisite, allowing changes to the implementation of part of the overall algorithm without involving modifications in the other parts, making the maintenance and extension easier.

The programming language *C++*, given its widespread use in the Optimisation community, was the one chosen for coding *ADDing*.

In this first full cycle (from the analysis phase to the test phase) of the development of

*ADDing*, we concentrated on the accomplishment of the first pre-requisite, while always having in mind the second one.

### 6.3.2 Main classes

In Figure 6.5, a simplified UML (Unified Modelling Language) class diagram of *ADDing* is depicted. After commenting on that simplified diagram, we will point out some classes and relations that were excluded. We designate classes with a capital letter and the objects that instantiate them in lower case.

The classes *Decomposition Model* and *Parameters* can be seen as inputs for the *Branch and Price* class that coordinates the branch-and-price algorithm, sending the output to the *Results* class. These one-to-one associations between those classes are denoted by the lines without arrows.

The class *Branch and Price* is composed (black diamond) of *Node Solver* and *Tree*. The *Node Solver* class is responsible for solving a node of the search tree. For that purpose, it is composed by a *Column Generation* class that has an association with *Subproblem*. This last class, as detailed in the previous Section, is the base class of an inheritance relation with *MySubproblem*.

An object of the class *Tree* is composed by several objects of the class *Node* (each associated with a node of the search tree). In addition, the class *Tree* has an association with the class *Subproblem* in order to generate branches.

The *Subproblem* class is associated with two very different classes. We chose that design option, for a practical reason: it allows the "tool-box" user to deal with only one class (apart from, of course, the *Decomposition Model* class).

The implementation of *ADDing* has a major difference in relation to the class diagram of Figure 6.5. In fact, the *Decomposition Model* class is composed by one *Subproblem* class. The same practical reason justifies that design option: in this way, a "black-box" user deals with only the class *Decomposition Model*.

One important class is not depicted in Figure 6.5: the one which represents an *Original Solution*. This class has associations with almost all the others. Along with a *Node* object (whose entire set of associatios are also not depicted in the simplified UML class diagram), it is responsible for the main flow of information between the classes.

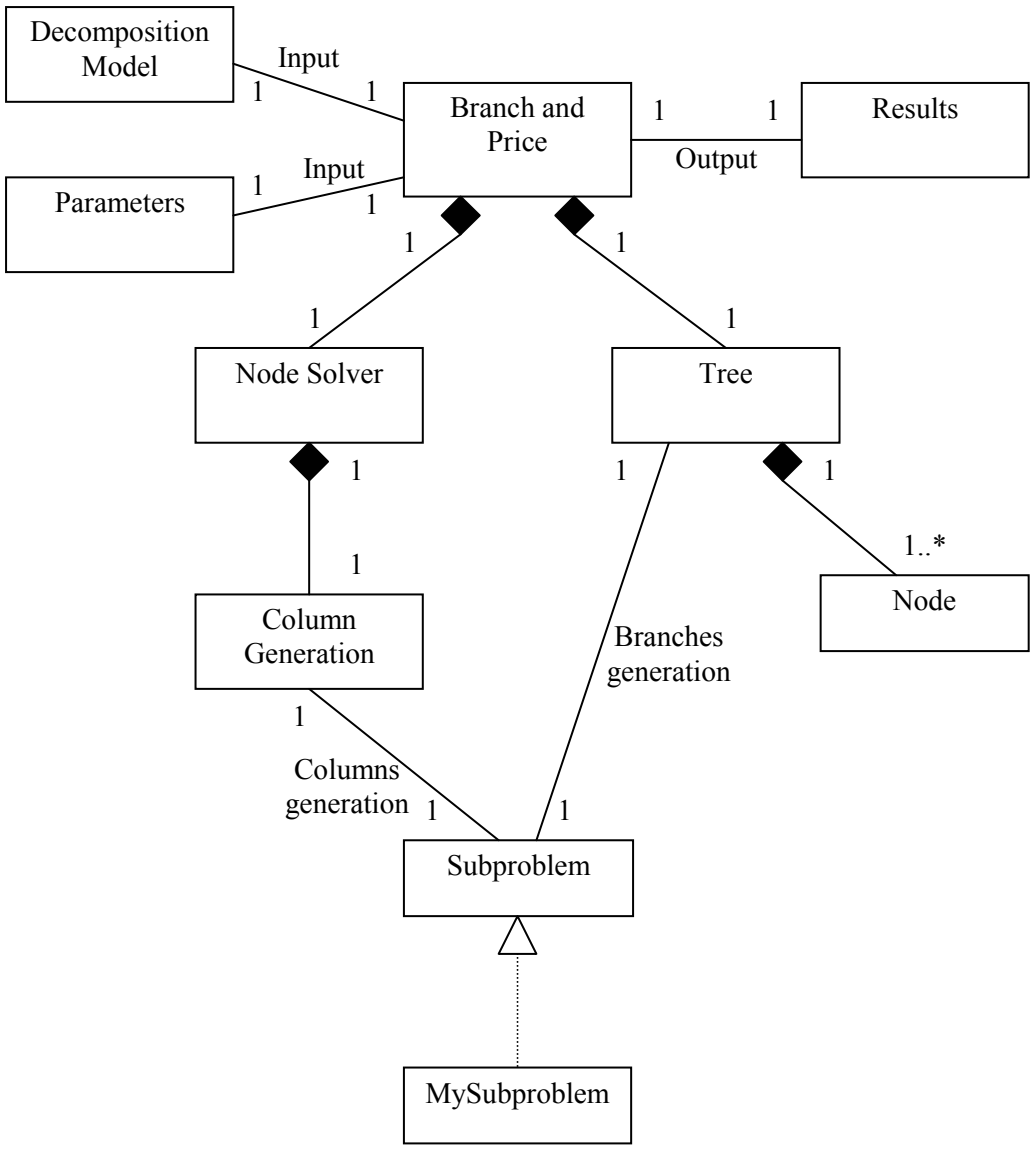



Figure 6.5 UML class diagram of *ADDing*.

### 6.3.3 Information flow

Two objects are responsible for the main flow of information, allowing the interaction of the other objects in order to implement the branch-and-price algorithm. Those two objects, namely *original solution* and *actual node* (class *Node*) are passed between public member functions of the classes that require them and modified through their own public member functions.

We now briefly describe their "journey" following a normal execution of a branch-and-price algorithm, giving some details on how the other classes deal with them.

The *actual node* object essentially contains the branching constraints of the node to be optimised, the lower bound given by its father, and (after being optimised) its own lower bound. The *original solution* object contains the information about the solution associated with the

node. Its flow is similar to the one of *actual node*.

It is up to the *tree* object to set the node to be optimised, returning it to *branch and price* when its function *GetNode()* is called. Then *branch and price* passes it down to *node solver*, which in turn passes the information relevant for the column generation algorithm (that is, the branching constraints) to *column generation*. This last object returns the result of the optimisation of the node and, if a feasible solution is obtained, updates *original solution*.

About the *Column Generation* class it is worth noting that the master model depicted in Figure 6.2 is not explicitly considered. All constraints (convexity, linking, and branching) are represented in the original variables in an object of a class *Constraint* (which distinguishes its type by an integer code) and has a pointer to the index of the corresponding row in the LP solver (currently only Cplex can be used). A similar representation is done for columns.

After the optimisation of the node, *actual node* and *original solution* make the inverse path, up to the *branch and price* object, which sends them along with the result of the optimisation (an integer) to the *tree* object. The *tree* private data structures are updated (generation of new nodes, modifying the incumbent value, ...) and the overall process is repeated until the *tree* states to *branch and price* that there are no more nodes to optimise.

In the current version of *ADDing*, the *Node Solver* class may seem redundant. It is included to allow the possible extension of *ADDing* in two directions: (i) by adding cuts and (ii) by allowing to solve a node by different solution methods.

In fact, a *Cut Generation* class (for lifted cover inequalities) at the same level as *Column Generation* in the UML class diagram was already implemented and tested in a specific problem. However, in order to belong to *ADDing*, more generality and simplicity of use (our main purposes) must be achieved with that class.

## 6.4 Conclusions

In this Chapter we presented *ADDing* (*Automatic Dantzig-Wolfe Decomposition for INteger column Generation*), a set of C++ classes that implements a branch-and-price algorithm, based on decomposing a (mixed) integer programming model.

Our main goal when developing *ADDing* was to provide its users with a simple and fast way to implement decomposition approaches for solving (mixed) integer problems. That goal was clearly obtained. When using *ADDing* as a "black-box", an original user-provided model is the only input required for *ADDing* to perform a decomposition and to obtain an optimal (integer) solution using branch-and-price. A main feature of *ADDing* is that the user can dynamically manage rows (in a transparent way), which may considerably improve the

efficiency of branch-and-price / column generation.

*ADDing* can also be used as a "tool-box". In that case, the user may implement specific subproblem solvers (including heuristic ones) and specific branching rules. That kind of utilisation involves a deeper knowledge of C++, but the few functions that have to be implemented in the derived class are expected to be easy to understand.

The internal structure of *ADDing* is based on the exchange of information between the original model and the master model. Keeping the branching constraints, derived in the original variables, in the model made the general – but simple – use that we intended easier.

At this time, the first cycle of the development of *ADDing* is finished. It has been used as a "tool-box" for two different decompositions for the binary multicommodity flow. All the features of *ADDing* here described were tested. It has also been used as a "black-box" for two different decompositions for a multi-item lot sizing with setup times problem.

We plan to improve *ADDing* further, in a near future, with two features already at an experimental phase: the incorporation of lifted cover inequalities and the combination of column generation and subgradient optimisation. In addition, there are two issues that can improve significantly the usability of *ADDing*: allowing the use of different LP solvers (which can be done by interfacing *ADDing* with COIN) and exploring the possibility of specifying the original model with a high-level / modelling language.

We definitely plan to explore other ideas. Firstly, multiple Dantzig-Wolfe decomposition. Secondly, hybridisation of branch-and-price and heuristics.

Having in mind that we must keep what we believe to be the main characteristic of *ADDing* – its simplicity of use –, those are challenging tasks.

# 7 General Conclusions

In the present Thesis, we presented column generation based algorithms for the linear, general integer, and binary minimum cost multicommodity flow problems.

Our general approach for (mixed) integer problems is based on using the Dantzig-Wolfe decomposition in a compact (original) formulation, combining column generation and branch-and-bound by defining the branching constraints on the original variables and keeping them in the (restricted) master model.

This approach was applied in the integer and in the binary multicommodity flow problems using a path decomposition. For the binary multicommodity flow problem, we also presented a decomposition based on defining the subproblem as a set of independent binary knapsack problems, which gives better quality lower bounds.

For the linear multicommodity flow problem defined in a planar network, we proposed a new model that allows the column generation approach to be significantly accelerated.

We detailed how to deal with negative cost cycles when using column generation for solving the path based decompositions of the three different multicommodity flow problems. In the case of the integer and the binary problems that is an important issue, because of the branching constraints of the type "greater than or equal to". In the case of the approach presented for accelerating column generation for the linear multicommodity flow problem in planar networks, negative cost cycles may appear because of the extra constraints used in the extended model.

Comparative computational results with a general-purpose solver were given for all the developed algorithms.

For the integer multicommodity flow problem, we used a set of instances publicly available that have been tested before by several other authors using methods for the linear multicommodity flow problem. The conclusions from those computational tests were not as expressive as we expected, since the linear relaxation of almost all the instances tested had an integral optimal solution. Anyway, the proposed algorithm provided better time results in several instances, and, for the larger ones, it can be concluded that it is the only feasible

direction to be followed, given the huge memory requirements of the formulation that must be used by a general-purpose solver (the one used in this work was Cplex 6.6).

For the binary multicommodity flow problem, our branch-and-price algorithm for the path decomposition provided slightly better results than the one previously developed in (Barnhart et al., 2000) (the main difference between the two algorithms is the branching strategy). Even with the use of general lifted cover inequalities in that algorithm (branch-and-price-and-cut) the results obtained were not competitive with the ones of the general-purpose solver Cplex 8.1, except for a few instances. The same happened with the branch-and-price algorithm based on the knapsack decomposition. Although giving better lower bounds, this decomposition proved to be particularly difficult to solve. Even using dynamic insertion and removal of rows, the majority of the larger instances tested could not be solved in one hour. To our best knowledge, the comparison between decomposition approaches and a general-purpose solver for the binary multicommodity flow problem was made for the first time. Given the fact that, in the literature, that approach is always taken as non-promising, this result came as a surprise.

For the linear multicommodity flow problem defined in a planar network, the proposed method for accelerating column generation was significantly faster than standard column generation in the randomly generated instances (some of them generated by other authors). For all instances that took more than two seconds to be solved by standard column generation, the relative improvement was always greater than 35% and frequently greater than 60%.

*ADDing* (*Automatic Dantzig-Wolfe Decomposition for INteger column Generation*), a set of *C++* classes, was developed. Its main purpose is to provide a simple and fast way to implement decomposition approaches for solving integer programming models by branch-and-price. The main distinctive feature of *ADDing*, when compared with the existing frameworks, is that it can be used as a "black-box": all that the user is required to do is to provide an original (mixed) integer model. It includes several features, in a transparent way to the user, that are time-consuming (and non-trivial) tasks when programming column generation based algorithms, such as the dynamic management of rows.

*ADDing* was used only to implement the decompositions and branch-and-price algorithms for the binary multicommodity flow problem, since its development was undertaken after the implementation of the algorithms for the other two problems.

Several issues were left open in this work.

We presented multiple Dantzig-Wolfe decomposition but did not implement it. In addition, we described how general cuts can be incorporated in branch-and-price, as long as they are based on the original variables, but only implemented lifted cover inequalities for a

specific decomposition. We plan to work on those issues in a near future.

The general-purpose solver we used (Cplex) was extremely efficient in solving a large number of instances of the problems we treated. However, decomposition approaches have clear advantages (at least, allowing the derivation of better lower bounds). We used a heuristic that consisted in solving a restricted master problem with Cplex at the end of the optimisation of the root node. We intend to explore further that kind of combination of column generation and a general-purpose integer programming solver.

The generality of the branch-and-price methodology presented here allows its extension to any (mixed) integer problem. Its implementation for other problems may further contribute to clarify the practical advantages and disadvantages of decomposition approaches when compared with state-of-the-art general-purpose solvers. *ADDing* may play an important role in that kind of comparison.

We intend to further explore column generation stabilisation methods, in particular for the knapsack decomposition of the binary multicommodity flow problem and by extending the stabilisation approach that was successfully used for the planar multicommodity flow problem to other network flow problems.

# References


Aardal, K., Weismantel, R. and Wolsey, L. A. (2002), *Non-standard approaches to integer programming*, Discrete Applied Mathematics, 123, 5-74.

Aggarwal, A. K., Oblak, M. and Vemuganti, R. R. (1995), *A heuristic solution procedure for multicommodity integer flows*, Computers and Operations Research, 22, 1075-1087.

Ahuja, R. K., Magnanti, T. L. and Orlin, J. B. (1993) *Network Flows,* Prentice Hall, Englewood Cliffs, NJ.

Akker, J. M., Hurkens, C. A. J. and Savelsbergh, M. W. P. (2000), *Time-indexed formulations for single-machine scheduling problems: column generation*, INFORMS Journal on Computing, 12, 111-124.

Akker, J. M. v. d., Hoogeveen, J. A. and Velde, S. L. v. d. (1999), *Parallel machine scheduling by column generation*, Operations Research, 47, 862-872.

Akker, M. v. d., Hoogeveen, H. and Velde, S. v. d. (2002), *Combining column generation and lagrangean relaxation to solve a single machine common due date problem*, INFORMS Journal on Computing, 14, 37-51.

Ali, A., Helgason, R., Kennington, J. and Lall, H. (1980), *Computational comparison among three multicommodity network flow algorithms*, Operations Research, 28, 995-1000.

Alvelos, F. (2005), *Multicommodity flows*, webpage: *http://www.dps.uminho.pt/pessoais/falvelos/research/index.html* (available on 26 January 2005).

Alvelos, F. and Carvalho, J. M. V. d. (2003), *Comparing branch-and-price algorithms for the unsplittable multicommodity flow problem*, In proceedings of the INOC - International Network Optimization Conference, Ed. Ben-Ameur, W. and Petrowski, A., Evry/Paris, October 2003, pp. 7-12.

Alves, C. and Carvalho, J. M. V. d. (2003), *A stabilized branch-and-price algorithm for integer variable sized bin-packing problems*, Universidade do Minho, Cadernos do DPS 14/2003.

Amor, H. B., Desrosiers, J. and Carvalho, J. M. V. d. (2003), *Dual-optimal inequalities for stabilized column generation*, GERAD, Les Cahiers du GERAD G-2003-20.

Aragão, M. P. d. and Uchoa, E. (2003), *Integer program reformulation for robust branch-and-cut-and-price algorithms*, working paper available at *http://www.inf.puc-rio.br/~uchoa/doc/cvpub.html* on 26 January 2005.

Assad, A. A. (1978), *Multicommodity network flows - A survey*, Networks, 8, 37-91.

Bahiense, L., Maculan, N. and Sagastizábal, C. (2002), *The volume algorithm revisited: relation with bundle methods*, Mathematical Programming, 94, 41-69.

Barahona, F. and Anbil, R. (2000), *The volume algorithm: producing primal solutions with a subgradient method*, Mathematical Programming, 87, 385-399.

Bard, J. F. and Purnomo, H. W. (2004), *Preference scheduling for nurses using column generation*, European Journal of Operational Research, 164, 510-534.

Barnhart, C. (1993), *Dual-ascent methods for large-scale multicommodity flow problems*, Naval Research Logistics, 40, 305-324.

Barnhart, C., Hane, C. A., Johnson, E. L. and Sigismondi, G. (1995), *A column generation and partitioning approach for multicommodity flow problems*, Telecommunications Systems, 3, 239-258.

Barnhart, C., Hane, C. A. and Vance, P. H. (2000), *Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems*, Operations Research, 48, 318-326.

Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P. and Vance, P. H. (1998), *Branch-and-price: column generation for solving huge integer programs*, Operations Research, 46, 316-329.

Barnhart, C. and Schneur, R. R. (1996), *Air network design for express shipment service*, Operations Research, 44, 852-863.

Barnhart, C. and Sheffi, Y. (1993), *A network-based primal-dual heuristic for the solution of multicommodity network flow problems*, Transportation Science, 27, 102-117.

Bazaraa, M. S. and Jarvis, J. J. (1977) *Linear Programming and Network Flows,* John Wiley and Sons.

Bazaraa, M. S., Sherali, H. D. and Shetty, C. M. (1993) *Nonlinear Programming - Theory and Algorithms,* John Wiley and Sons.

Beasley, J. E. (1995), *Lagrangean relaxation*, In *Modern Heuristic Techniques for Combinatorial Problems*, Ed. Reeves, C., McGraw Hill.

Behzad, M., Chartrand, G. and Lesniak-Foster, L. (1979) *Graphs & Digraphs,* Wadsworth.

Belaidouni, M. and Ben-Ameur, W. (2003), *A superadditive approach to solve the minimum cost single path routing problem: preliminary results*, In proceedings of the INOC - International Network Optimization Conference, Ed. Ben-Ameur, W. and Petrowski, A., Evry/Paris, October 2003, pp. 67-71.

Benders, J. F. (1962), *Partitioning procedures for solving mixed-variables programming problems*, Numerische Mathematik, 4, 238-252.

Bertsekas, D. (1999) *Nonlinear programming,* Athena Scientific, Belmont, Massachusetts.

Bixby, R. E., Fenelon, M., Gu, Z., Rothberg, E. and Wunderling, R. (2000), *MIP: theory and practice - closing the gap*, In *System Modelling and Optimization Methods, Theory and Applications*, Ed. Powell, M. J. D. and Scholtes, S., Kluwer, pp. 19-49.

Boland, N., Hamacher, H. W. and Lenzen, F. (2004), *Minimizing beam-on time in cancer radiation treatment using multileaf collimators*, Networks, 43, 226-240.

Boland, N. and Surendonk, T. (2001), *A Column Generation Approach to Delivery Planning over Time with Inhomogeneous Service Providers and Service Interval Constraints*, Annals of Operations Research, 108, 143-156.

Bourjolly, J.-M., Laporte, G. and Mercure, H. (1997), *A combinatorial column generation algorithm for the maximum stable set problem*, Operations Research Letters, 20, 21-29.

Bredström, D., Lundgren, J. T., Rönnqvist, M., Carlsson, D. and Mason, A. (2004), *Supply chain optimization in the pulp mill industry - IP models, column generation and novel constraint branches*, European Journal of Operational Research, 156, 2-22.

Briant, O., Lemaréchal, C., Meurdesoif, P., Michel, S., Perrot, N. and Vanderbeck, F. (2004), *Comparison of bundle and classical column generation*, INRIA, research report, available at *http://www.inria.fr/rrrt/rr-5453.html* on 26 January 2005.

Butt, S. E. and Ryan, D. M. (1999), *An optimal solution procedure for the multiple tour maximum collection problem using column generation*, Computers & Operations Research, 26, 427-441.

Cappanera, P. and Frangioni, A. (2000), *Embedding a bundle method in a branch and bound framework: an application-oriented development*, Università di Pisa, Dipartimento di Informatica, TR-00-09.

Cappanera, P. and Frangioni, A. (2003), *Symmetric and asymmetric parallelization of a cost-decomposition algorithm for multi-commodity flow problems*, INFORMS Journal on Computing, 15, 369 -384.

Caprara, A., Lancia, G. and Ng, S.-K. (2001), *Sorting permutations by reversals through branch-and-price*, INFORMS Journal on Computing, 13, 224-244.

Carolan, W. J., Hill, J. E., Kennington, J. K., Niemi, S. and Wichmann, S. J. (1990), *An empirical evaluation of the KORBX algorithms for military airlift applications*, Operations Research, 38, 240-248.

Carvalho, J. M. V. d. (1998), *Exact solution of one-dimensional cutting stock problems using column generation and branch-and-bound*, International Transactions in Operational Research, 5, 35-44.

Carvalho, J. M. V. d. (1999), *Exact solution of bin-packing problems using column generation and branch-and-bound*, Annals of Operations Research, 86, 629-659.

Carvalho, J. M. V. d. (2000), *Using extra dual cuts to accelerate column generation*, to appear in INFORMS Journal on Computing.

Castro, J. (2000), *A specialized interior-point algorithm for multicommodity network flows*, SIAM Journal on Optimization, 10, 852-877.

Castro, J. and Frangioni, A. (2000), *A paralell implementation of an interior-point algorithm for multicommodity network flows*, Universitat Politècnica de Catalunya, DR2000-06.

Castro, J. and Nabona, N. (1994), *PPRN 1.0, User's Guide*, available at *http://www-eio.upc.es/~jcastro* on 26 January 2005.

Castro, J. and Nabona, N. (1996), *An implementation of linear and nonlinear multicommodity network flows*, European Journal of Operational Research, 92, 37-53.

Ceselli, A. and Righini, G. (2002), *A branch and price algorithm for the capacitated p-median problem*, to appear in Networks.

Chardaire, P. and Lisser, A. (2002a), *Minimum-cost multicommodity flows*, In *Handbook of Applied Optimization*, Ed. Pardalos, P. M. and Resende, M. G. C., Oxford University Press, pp. 404-422.

Chardaire, P. and Lisser, A. (2002b), *Simplex and interior point specialized algorithms for solving nonoriented multicommodity flow problems*, Operations Research, 50, 260-276.

Chen, Z.-L. and Powell, W. B. (1999), *A column generation based decomposition algorithm for a parallel machine just-in-time scheduling problem*, European Journal of Operational Research, 116, 220-232.

Christiansen, M. and Fagerholt, J. (2002), *Robust ship scheduling with multiple time windows*, Naval Research Logistics, 49, 611-625.

Christiansen, M. and Nygreen, B. (1998), *Modelling path flows for a combined ship routing and inventory management problem*, Annals of Operations Research, 82, 391-412.

Costa, M.-C., Hertz, A. and Mittaz, M. (2002), *Bounds and heuristics for the shortest capacitated paths problem*, Journal of Heuristics, 8, 449-465.

Crainic, T. G., Frangioni, A. and Gendron, B. (2001), *Bundle-based relaxation methods for multicommodity capacitated fixed charge network design*, Discrete Applied Mathematics, 112, 73-99.

Crowder, H. P., Johnson, E. L. and Padberg, M. W. (1983), *Solving large scale 0-1 linear programming problems*, Operations Research, 31, 803-834.

Dantzig, G. B. (1963) *Linear Programming and Extensions,* Princeton University Press, Princeton, New Jersey.

Dantzig, G. B. and Wolfe, P. (1960), *Decomposition principle for linear programs*, Operations Research, 8, 101-111.

Degraeve, Z. and Jans, R. (2003), *Improved lower bounds for the capacitated lot sizing problem with set up times*, Erasmus Research Institute of Management, ERS-2003-026-LIS.

Desaulniers, G., Desrosiers, J. and Solomon, M. M. (2001), *Accelerating strategies in column generation methods for vehicle routing and crew scheduling problems*, In *Essays and surveys in metaheuristics*, Ed. Ribeiro, C. C. and Hansen, P., Kluwer, pp. 309-324.

Desaulniers, G., Lavigne, J. and Soumis, F. (1998), *Multi-depot vehicle scheduling problems with time windows and waiting costs*, European Journal of Operational Research, 111, 479-494.

Desrochers, M., Desrosiers, J. and Solomon, M. (1992), *A new optimization algorithm for the vehicle routing problem with time windows*, Operations Research, 40, 342-354.

Desrochers, M. and Soumis, F. (1989), *A column generation approach to the urban transit crew scheduling problem*, Transportation Science, 23, 1-13.

Desrosiers, J., Dumas, Y., Solomon, M. M. and Soumis, F. (1995), *Time constrained routing and scheduling*, In *Network Routing*, Ed. al., M. O. B. e., Elsevier Science, Amsterdam, pp. 35-139.

Desrosiers, J., Soumis, F. and Desrochers, M. (1984), *Routing with time windows by column generation*, Networks, 14, 545-565.

Detlefsen, N. K. and Wallace, S. W. (2002), *The simplex algorithm for multicommodity networks*, Networks, 39, 15-28.

Ebem-Chaime, M., Tovey, C. A. and Ammons, J. C. (1996), *Circuit partitioning via set partitioning and column generation*, Operations Research, 44, 65-76.

Elhedhli, S. and Goffin, J.-L. (2001), *The integration of an interior-point cutting-plane method within a branch-and-price algorithm*, GERAD, Les Cahiers du GERAD G-2001-19.

Evans, J. R. (1977), *Some network flow models and heuristics for multiproduct production and inventory planning*, AIIE Transactions, 75-81.

Eveborn, P. and Rönnqvist, M. (2004), *Scheduler - a system for staff planning*, Annals of Operations Research, 128, 21-45.

Fahle, T., Junker, U., Karisch, S. E., Kohl, N., Sellmann, M. and Vaaben, B. (2002), *Constraint Programming Based Column Generation for Crew Assignment.*, Journal of Heuristics, 8, 59-81.

Farvolden, J. M., Powell, W. B. and Lustig, I. J. (1993), *A primal partitioning solution for the arc-chain formulation of a multicommodity network flow problem*, Operations Research, 41, 669-693.

Fisher, M. L. (1981), *The lagrangian relaxation method for solving integer programming problems*, Management Science, 27, 1-18.

Fleischer, L. K. (2000), *Approximating Fractional Multicommodity Flow Independent of the Number of Commodities*, SIAM Journal on Discrete Mathematics, 13, 505-520.

Ford, L. R. and Fulkerson, D. R. (1958), *A suggested computation for maximal multicommodity network flows*, Management Science, 5, 97-101.

Ford, L. R. and Fulkerson, D. R. (1962) *Flows in Networks,* Princeton University Press.

Frangioni, A. (1997), *Dual-Ascent Methods and Multicommodity Flow Problems*, Ph. D. Thesis, Dipartimento di Informatica, Università di Pisa-Genova-Udine.

Frangioni, A. (2002), *Generalized Bundle Methods*, SIAM Journal on Optimization, 13, 117-156.

Frangioni, A. (2004), *About lagrangian methods in integer optimization*, to appear in Annals of Operations Research.

Frangioni, A. (2005), *Multicommodity flow problems*, webpage: *http://www.di.unipi.it/di/groups/optimize/Data/MMCF.html* (available on 26 January 2005).

Frangioni, A. and Gallo, G. (1999), *A bundle type dual-ascent approach to linear multicommodity min cost flow problems*, INFORMS Journal on Computing, 11, 370-393.

Freling, R., Huisman, D. and Wagelmans, A. P. M. (2003), *Models and algorithms for integration of vehicle and crew scheduling*, Journal of Scheduling, 6, 63-85.

Gallo, G. and Pallottino, S. (1988), *Shortest path algorithms*, Annals of Operations Research, 13, 3-79.

Gamache, M., Soumis, F., Marquis, G. and Desrosiers, J. (1999), *A column generation approach for large-scale aircrew rostering problems*, Operations Research, 47, 247-263.

Garey, M. R. and Johnson, D. S. (1979) *Computers and Intractability, A Guide to the Theory of NP-Completeness,* W. H. Freeman and Company.

Gendron, B., Crainic, T. G. and Frangioni, A. (1999), *Multicommodity capacitated network design*, In *Telecommunications Network Planning*, Ed. Soriano, P. and Sansòn, B., Kluwer Academic Publisher, pp. 1-19.

Geoffrion, A. M. (1974), *Lagrangean relaxation for integer programming*, Mathematical Programming Study, 2, 82-114.

Gilmore, P. C. and Gomory, R. E. (1961), *A linear programming approach to the cutting stock problem*, Operations Research, 9, 849-859.

Gilmore, P. C. and Gomory, R. E. (1963), *A linear programming approach to the cutting stock problem - part II*, Operations Research, 11, 863-888.

Goffin, J. L., Gondzio, J., Sarkissian, R. and Vial, J. P. (1996), *Solving nonlinear multicommodity flow problems by the analytic center cutting plane method*, Mathematical Programming, 76, 131-154.

Goffin, J. L., Haurie, A. and Vial, J. P. (1992), *Decomposition and nondifferentiable optimization with the projective algorithm*, Management Science, 38, 284-302.

Goffin, J.-L., Haurie, A., Vial, J.-P. and Zhu, D. L. (1993), *Using central prices in decomposition of linear programs*, European Journal of Operational Research, 64, 393-409.

Goffin, J.-L. and Vial, J.-P. (1999), *Convex nondifferentiable optimization: a survey focussed on the analytic center cutting plane method*, HEC/Logilab, Technical Report 99.02.

Goldberg, A. V., Oldham, J. D., Plotkin, S. and Stein, C. (1998), *An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow*, In proceedings of the 6th International IPCO Conference, Ed. Bixby, R. E., Boyd, E. A. and Ríos-Mercado, R. Z., Houston, Texas, June 1998, pp. 338-352.

Gu, Z., Nemhauser, G. L. and Savelsbergh, M. W. P. (1998), *Lifted cover inequalities for 0-1 integer programs: computation*, INFORMS Journal on Computing, 10, 427-437.

Gu, Z., Nemhauser, G. L. and Savelsbergh, M. W. P. (1999), *Lifted cover inequalities for 0-1 integer programs: complexity*, INFORMS Journal on Computing, 11, 117-123.

Guignard, M. and Kim, S. (1987), *Lagrangean decomposition: a model yielding stronger lagrangean bounds*, Mathematical Programming, 39, 215-228.

Günlük, O., Ladanyi, L. and Vries, S. d. (2002), *A branch-and-price algorithm and new test problems for spectrum auctions*, IBM, research report RC22530.

Haase, K., Desaulniers, G. and Desrosiers, D. (2001), *Simultaneous vehicle and crew scheduling in urban mass transit systems*, Transportation Science, 35, 286-303.

Hane, C. A., Barnhart, C., Johnson, E. L., Marsten, R. E., Nemhauser, G. L. and Sigismondi, G. (1995), *The fleet assignment problem: solving a large-scale integer program*, Mathematical Programming, 70, 211-232.

Held, M. and Karp, R. M. (1970), *The traveling-salesman problem and minimum spanning trees*, Operations Research, 18, 1138-1167.

Held, M. and Karp, R. M. (1971), *The traveling-salesman problem and minimum spanning trees: Part II*, Mathematical Programming, 1, 6-25.

Held, M., Wolfe, P. and Crowder, H. P. (1974), *Validation of subgradient optimization*, Mathematical Programming, 6, 62-88.

Henningsson, M., Holmberg, K., Rönnqvist, M. and Värbrand, P. (2002), *Ring network design by lagrangean based column generation*, Telecommunications Systems, 21, 301-318.

Hiriart-Urruty, J.-B. and Lemaréchal, C. (1993a) *Convex analysis and minimisation algorithms I,* Springer-Verlag.

Hiriart-Urruty, J.-B. and Lemaréchal, C. (1993b) *Convex analysis and minimisation algorithms II,* Springer-Verlag.

Hoffman, K. L. (2000), *Combinatorial optimization: Current successes and directions for the future*, Journal of Computational and Applied Mathematics, 124, 341-360.

Holmberg, K. and Yuan, D. (2000), *A Lagrangean heuristic based branch-and-bound approach for the capacitated network design problem*, Operations Research, 48, 461-481.

Holmberg, K. and Yuan, D. (2001), *A multicommodity network flow problem with side constraints on paths solved by column generation*, Department of Mathematics, Linköpings Universitet, in Linköping Studies in Science and Technologie, Dissertations No. 682.

Hopcroft, J. and Tarjan, R. (1974), *Efficient planarity testing*, Journal of the ACM, 21, 549-568.

Horowitz, E. and Sahni, S. (1974), *Computing partitions with applications to the knapsack problem*, Journal of the ACM, 21, 277-292.

Hu, T. C. (1963), *Multicommodity network flows*, Operations Research, 11, 344-360.

ILOG (1999) *CPLEX 6.5, User's Manual*.

ILOG (2002) *CPLEX 8.0, User's Manual*.

Jaumard, B., Hansen, P. and Aragão, M. P. d. (1991), *Column generation methods for probabilistic logic*, ORSA Journal on Computing, 3, 135-148.

Jaumard, B., Marcotte, O., Meyer, C. and Vovor, T. (2002), *Erratum to "Comparison of column generation models for channel assignment in cellular networks"*, Discrete Applied Mathematics, 118, 299-322.

Jaumard, B., Semet, F. and Vovor, T. (1998), *A generalized linear programming model for nurse scheduling*, European Journal of Operational Research, 107, 1-18.

Jeong, G., Lee, K., Park, S. and Park, K. (2002), *A branch-and-price algorithm for the Steiner tree packing problem*, Computers and Operations Research, 29, 221-241.

Johnson, E. L., Nemhauser, G. L. and Savelsbergh, M. W. P. (2000), *Progress in linear programming based branch-and-bound algorithms: an exposition*, INFORMS Journal on Computing, 12, 2-23.

Jones, K. L., Lustig, I. J., Farvolden, J. M. and Powell, W. B. (1993), *Multicommodity network flows: The impact of formulation on decomposition*, Mathematical Programming, 62, 95-117.

Jørgensen, D. G. and Meyling, M. (2002), *A branch-and-price algorithm for switch-box routing*, Networks, 40, 13-26.

Jünger, M. and Thienel, S. (2000), *The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization*, Software - Practice & Experience, 30.

Kallehauge, B., Larsen, J. and Madsen, O. B. G. (2001), *Lagrangean duality applied on vehicle routing with time windows*, Technical University of Denmark, IMM-TR-2001-9.

Kang, S., Malik, K. and Thomas, L. J. (1999), *Lotsizing and scheduling on parallel machines with sequence-dependent setup costs*, Management Science, 45, 272-289.

Kapoor, S. and Vaidya, P. M. (1996), *Speeding up Karmarkar's algorithm for multicommodity flows*, Mathematical Programming, 73, 111-127.

Kelley, J. E. (1960), *The cutting-plane method for solving convex programs*, Journal of the SIAM, 8, 703-712.

Kennington, J. and Shalaby, M. (1977), *An effective subgradient procedure for minimal cost multicommodity flow problems*, Management Science, 23, 994-1004.

Kennington, J. L. (1978), *Survey of linear cost multicommodity network flows*, Operations Research, 26, 209-236.

Kennington, J. L. and Helgason, R. V. (1980) *Algorithms for network programming,* Wiley, New York.

Klingman, D., Napiers, A. and Stutz, J. (1974), *NETGEN: A program for generating large scale capacitated assignment, tranportation, and minimum cost flow network problems*, Management Science, 20, 814-821.

Klose, A. and Drexl, A. (2002), *A partitioning and column generation approach for the capacitated facility location problem*, University of St. Gallen, working paper available at *http://www.wiwi.uni-wuppertal.de/Publikatio.626.0.html* on 26 January 2005.

Kohl, N., Desrosiers, J., Madsen, O. B. G., Solomon, M. M. and Soumis, F. (1999), *2-path cuts for the vehicle routing problem with time windows*, Transportation Science, 33, 101-116.

Kolliopoulos, S. G. and Stein, C. (1999), *Experimental evaluation of approximation algorithms for single-source unsplittable flow*, In proceedings of the 7th International Integer Programming and Combinatorial Optimization Conference, Ed. Cornuéjols, G., Burkard, R. E. and Woeginger, G. J., Graz, Austria, pp. 328-344.

Larsson, T., Patriksson, M. and Rydergren, C. (2004), *A column generation procedure for the side constrained traffic equilibrium problem*, Transportation Research Part B, 38, 17-38.

Larsson, T. and Yuan, D. (2004), *An augmented lagrangian algorithm for large scale multicommodity routing*, Computational Optimization and Applications, 27, 187-215.

Lee, T. and Park, S. (2001), *An integer programming approach to the time slot assignment problem in SS/TDMA systems with intersatellite links*, European Journal of Operational

Research, 135, 57-66.

Lemaréchal, C. (1989), *Nondifferentiable optimization*, In *Optimization*, Ed. Nemhauser, G. L., Kan, A. H. G. R. and Todd, M. J., Elsevier Science, Amsterdam.

Lemaréchal, C. (2003), *The omnipresence of Lagrange*, 4OR Quarterly Journal of the Belgian, French and Italian Operations Research Societies, 1, 7-25.

Lemaréchal, C., Nemirovskii, A. and Nesterov, Y. (1995), *New variants of bundle methods*, Mathematical Programming, 69, 111-147.

Liebers, A. (2001), *Planarizing Graphs - A Survey and Annotated Bibliography*, Journal of Graphs Algorithms and Applications, 5, 1-74.

Lingaya, N., Cordeau, J.-F., Desaulniers, G., Desrosiers, J. and Soumis, F. (2002), *Operational car assignment at VIA Rail Canada*, Transportation Research Part B, 36, 755-778.

Lorena, L. A. N. and Senne, E. L. F. (2004), *A column generation approach to capacitated p-median problems*, Computers and Operations Research, 31, 863-876.

Lübbecke, M. E. and Desrosiers, J. (2002), *Selected topics in column generation*, GERAD, Les Cahiers de GERAD G-2002-64.

Lübbecke, M. E. and Zimmermann, U. T. (2003), *Engine routing and scheduling at industrial in-plant railroads*, to appear in Transportation Science, to appear.

Magnanti, T. L., Shapiro, J. F. and Wagner, M. H. (1976), *Generalized linear programming solves the dual*, Management Science, 22, 1195-1203.

Magnanti, T. L. and Wong, R. T. (1984), *Network design and transportation planning: Models and algorithms*, Transportation Science, 18, 1-55.

Mamer, J. W. and McBride, R. D. (2000), *A decomposition-based pricing procedure for large-scale linear programs: an apllication to the linear multicommodity flow problem*, Management Science, 46, 693-709.

Marchand, H., Martin, A., Weismantel, R. and Wolsey, L. (2002), *Cutting planes in integer and mixed integer programming*, Discrete Applied Mathematics, 123, 397-446.

Marsten, R. E., Hogan, W. W. and Blankenship, J. W. (1975), *The boxstep method for large-scale optimization*, Operations Research, 23, 389-405.

Martin, R. K. (1999) *Large Scale Linear and Integer Optimization, A Unified Approach,* Kluwer Academic Publishers.

Martins, I., Constantino, M. and Borges, J. G. (2003), *A column generation approach for solving a non-temporal forest harvest model with spatial structure constraints*, to appear in European Journal of Operational Research.

McBride, R. D. (1998), *Progress made in solving the multicommodity flow problem*, SIAM Journal on Optimization, 8, 947-955.

McBride, R. D. and Mamer, J. W. (1997), *Solving multicommodity flow problems with a primal embedded network simplex algorithm*, INFORMS Journal on Computing, 9, 154-163.

McBride, R. D. and Mamer, J. W. (2001), *Solving the undirected multicommodity flow problem*

*using a shortest path-based pricing algorithm*, Networks, 38, 181-188.

Medhi, D. (1994), *Bundle-based decomposition for large-scale convex optimization: Error estimate and application to block-angular linear programs*, Mathematical Programming, 66, 79-101.

Mehlhorn, K. and Näher, S. (1999) *LEDA - A Platform for Combinatorial and Geometric Computing,* Cambridge University Press, Cambridge.

Mehrotra, A., Murphy, K. E. and Trick, M. A. (2000), *Optimal shift scheduling: a branch-and-price approach*, Naval Research Logistics, 47, 185-200.

Mehrotra, A., Natraj, N. R. and Trick, M. A. (2001), *Consolidating maintenance spares*, Computational Optimization and Applications, 18, 251-272.

Mehrotra, A. and Trick, M. A. (1996), *A column generation approach for graph coloring*, INFORMS Journal on Computing, 8, 344-354.

Mehrotra, A. and Trick, M. A. (1998), *Cliques and clustering: A combinatorial approach*, Operations Research Letters, 22, 1-12.

Merle, O. d., Villeneuve, D., Desrosiers, J. and Hansen, P. (1999), *Stabilized column generation*, Discrete Mathematics, 194, 229-237.

Minoux, M. (1986) *Mathematical Programming Theory and Algorithms,* John Wiley and Sons.

Minoux, M. (1989), *Network synthesis and optimum network design problems: Models, solution methods and applications*, Networks, 19, 313-360.

Murty, K. G. (1983) *Linear Programming,* John Wiley and Sons.

Nemhauser, G. L. (1994), *The age of optimization: solving large-scale real-world problems*, Operations Research, 42, 5-13.

Nemhauser, G. L. and Wolsey, L. A. (1999) *Integer and Combinatorial Optimization,* John Wiley and Sons.

Ouaja, W. and Richards, B. (2003), *A hybrid solver for optimal routing of bandwidth-guaranteed traffic*, In proceedings of the International Network Optimization Conference, Ed. Ben-Ameur, W. and Petrowski, A., Evry - Paris, pp. 441-447.

Ouaja, W. and Richards, B. (2004), *A hybrid multicommodity routing algorithm for traffic engineering*, Networks, 43, 125-140.

Ouorou, A., Mahey, P. and Vial, J. P. (2000), *A survey of algorithms for convex multicommodity flow problems*, Management Science, 46, 126-147.

Padberg, M. W. and Rinaldi, G. (1991), *A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems*, SIAM Review, 33, 60-100.

Park, K., Kang, S. and Park, S. (1996), *An integer programming approach to the bandwidth packing problem*, Management Science, 42, 1277-1291.

Park, S., Kim, D. and Lee, K. (2003), *An integer programming approach to the path selection problems*, In proceedings of the International Network Optimization Conference, Ed. Ben-Ameur, W. and Petrowski, A., Evry - Paris, pp. 448-453.

Parker, M. and Ryan, J. (1994), *A column generation algorithm for bandwidth packing*, Telecommunications Systems, 2, 185-195.

Persson, J. A. and Göthe-Lundgren, M. (2005), *Shipment planning at oil refineries using column generation and valid inequalities*, European Journal of Operational Research, 163, 631-652.

Péton, O. and Vial, J.-P. (2001), *A tutorial on ACCPM: user's guide for version 2.01*, University of Geneva, HEC/Logilab.

Pimentel, C., Alvelos, F. and Carvalho, J. M. V. d. (2004), *A branch-and-price algorithm for the multi-item capacitated lot-sizing problem with setup times*, presented in Optimization 2004, Lisbon, July 2004.

Pinar, M. C. and Zenios, S. A. (1994), *A data-level parallel linear-quadratic penalty algorithm for multicommodity network flows*, ACM Transactions on Mathematical Software, 20, 531-552.

Ralphs, T. K. and Ladányi, L. (2001), *COIN/BCP User's Manual*, available at *http://www.coin-or.org/* on 26 January 2005.

Ralphs, T. K. and Ladányi, L. (2003), *SYMPHONY 4.0 User's Manual*, available at *http://www.branchandcut.org/SYMPHONY/* on 26 January 2005.

Ribeiro, C. C., Minoux, M. and Penna, M. C. (1989), *An optimal column-generation-with-ranking algorithm for very large scale set partitioning problems in traffic assignment*, European Journal of Operational Research, 41, 232-239.

Ribeiro, C. C. and Soumis, F. (1994), *A column generation approach to the multiple-depot vehicle scheduling problem*, Operations Research, 42, 41-52.

Rosen, J. B. (1964), *Primal partitioning programming for block diagonal matrices*, Numerische Mathematik, 6, 250-260.

Roy, T. J. v. (1986), *A cross decomposition algorithm for capacitated facility location*, Operations Research, 34, 145-163.

Sankaran, J. K. (1995), *Column generation applied to linear programs in course registration*, European Journal of Operational Research, 87, 328-342.

Sarin, S. C. and Aggarwal, S. (2001), *Modeling and algorithmic development of a staff scheduling problem*, European Journal of Operational Research, 128, 558-569.

Savelsbergh, M. (1997), *A branch-and-price algorithm for the generalized assignment problem*, Operations Research, 45, 831-841.

Savelsbergh, M. and Sol, M. (1998), *DRIVE: Dynamic routing of independent vehicles*, Operations Research, 46, 474-490.

Saviozzi, G. (1986), *Advanced start for multicommodity network flow problem*, Mathematical Programming Study, 26, 221-224.

Schneur, R. R. and Orlin, J. B. (1998), *A scaling algorithm for the multicommodity flow problems*, Operations Research, 46, 231-246.

Schultz, G. L. and Meyer, R. R. (1991), *An interior point method for block angular*

*optimization*, SIAM Journal on Optimization, 1, 583-602.

Senne, E. L. F., Lorena, L. A. N. and Pereira, M. A. (2005), *A branch-and-price approach to p-median location problems*, Computers and Operations Research, 32, 1655-1664.

Shapiro, F. (1979), *A survey of lagrangean techniques for discrete optimization*, Annals of Discrete Mathematics, 5, 113-138.

Shaw, D. X. (1999), *A unified limited column generation approach for facility location problems on trees*, Annals of Operations Research, 87, 363-382.

Shetty, B. and Muthukrishnan, R. (1990), *A parallel projection for the multicommodity network model*, Journal of the Operational Research Society, 41, 837-842.

Soroush, H. and Mirchandani, P. B. (1990), *The stochastic multicommodity flow problem*, Networks, 20, 121-155.

Soumis, F. (1997), *Decomposition and column generation*, In *Annotated Bibliographies in Combinatorial Optimization*, Ed. Dell'Amico, M., Maffioli, F. and Martello, S., John Wiley and Sons.

Sutter, A., Vanderbeck, F. and Wolsey, L. (1998), *Optimal placement of add/drop multiplexers: heuristic and exact algorithms*, Operations Research, 46, 719-728.

Thienel, S. (1995), *ABACUS - A Branch-And-CUt System*, Ph. D. Thesis, University of Cologne.

Tombus, Ö. and Bilgiç, T. (2004), *A column generation approach to the coalition formation problem in multi-agent systems*, Computers and Operations Research, 31, 1635-1653.

Tomlin, J. A. (1966), *Minimum-cost multicommodity network flows*, Operations Research, 14, 45-51.

Vance, P. H. (1998), *Branch-and-price algorithms for the one-dimensional cutting stock problem*, Computational Optimization and Applications, 9, 211-228.

Vance, P. H., Barnhart, C., Johnson, E. L. and Nemhauser, G. L. (1994), *Solving binary cutting stock problems by column generation and branch-and-bound*, Computational Optimization and Applications, 3, 111-130.

Vance, P. H., Barnhart, C., Johnson, E. L. and Nemhauser, G. L. (1997), *Airline crew scheduling: A new formulation and decomposition algorithm*, Operations Research, 45, 188-200.

Vanderbeck, F. (1998), *Lot-sizing with start-up times*, Management Science, 44, 1409-1425.

Vanderbeck, F. (1999), *Computational study of a column generation algorithm for bin packing and cutting stock problems*, Mathematical Programming, 86, 565-594.

Vanderbeck, F. (2000), *On Dantzig-Wolde decomposition in integer programming and ways to perform branching in a branch-and-price algorithm*, Operations Research, 48, 111-128.

Vanderbeck, F. (2005), *Implementing mixed integer column generation*, In *Column Generation*, Ed. Desaulniers, G., Desrosiers, J. and Solomon, M. M., Kluwer, to appear.

Vanderbeck, F. and Wolsey, L. A. (1996), *An exact algorithm for IP column generation*,

Operations Research Letters, 19, 151-159.

Villeneuve, D., Desrosiers, J., Lübbecke, M. E. and Soumis, F. (2003), *On compact formulations for integer programs solved by column generation*, Technische Universitat Berlin, Institut für Mathematik, 2003/25.

Wang, Y. and Wang, Z. (1999), *Explicit routing algorithms for internet traffic engineering*, In proceedings of the International Conference on Computer Communication Networks, Ed. Boston, USA, pp. 582-588.

Wentges, P. (1997), *Weighted Dantzig-Wolfe decomposition for linear mixed-integer programming*, International Transactions in Operational Research, 4, 151-162.

Wilhelm, W. E. (1999), *A column-generation approach for the assembly system design problem with tool changes*, The International Journal of Flexible Manufacturing Systems, 11, 177-205.

Wilhelm, W. E. (2001), *A technical review of column generation in integer programming*, Optimization and Engineering, 2, 159-200.

Williams, H. P. (1999) *Model Building in Mathematical Programming,* John Wiley and Sons.

Wolfe, P. (1970), *Convergence theory in nonlinear programming*, In *Integer and Nonlinear Programming*, Ed. Abadie, J., North-Holland.

Wolsey, L. (2002), *Soving multi-item lot-sizing problems with a MIP solver using classification and reformulation*, Management Science, 48, 1587-1602.

Wolsey, L. A. (1998) *Integer Programming,* John Wiley and Sons.

Yan, S. and Chang, J.-C. (2002), *Airline cockpit crew scheduling*, European Journal of Operational Research, 136.

Yan, S., Tung, T.-T. and Tu, Y.-P. (2002), *Optimal construction of airline individual crew pairings*, Computers and Operations Research, 29, 341-363.

Yuan, D. (2001), *An annotated bibliography in communication network design and routing*, Department of Mathematics, Linköpings Universitet, in Linköping Studies in Science and Technologie, Dissertations No. 682.

Zahorik, A., Thomas, L. J. and Trigeiro, W. W. (1984), *Network programming models for production scheduling in multi-stage, multi-item capacitated systems*, Management Science, 30, 308-325.

# Appendix – *ADDing* Details

# *decmodel* functions

***Dimensions and information about variables***

*void SetDimensions (int NumSPs, int NumVars, int NumArts, int NumExtraVars)*

Sets the number of subproblems, the number of variables of each subproblem, the number of artificial variables to be used and the number of extra variables. Extra variables are linear variables that do not appear in the subproblem. Artificial variables can appear in the convexity and/or the original constraints of the master problem. The number of extra variables includes the number of artificials. Artificial variables are in the first *NumArts* positions of *NumExtraVars*. In the example of Figure 6.3, page 158 of the main text, the original model has 10 subproblems, each with 20 variables. The number of artificial variables to be used is 30. There are no extra variables.

*void SetTypeVars (int NumLinVars, int NumBinVars, int NumIntVars)*

Sets the number of linear variables, binary variables and general integer variables. The order is important: first *NumLinVars* are linear, the next *NumBinVars* are binary and the next *NumIntVars* are general integer ones. Note that *NumLinVars* + *NumBinVars* + *NumIntVars* must be equal to *aNumVars* defined in the previous method. In the example of Figure 6.3, page 158 of the main text, the original model has 10 binary variables.

*void SetObjVars (double ** ObjVars)*

Sets the coefficients of the variables in the objective function, *ObjVars[k][j]* contains the coefficient of the *(j+1)*-th variable of the *(k+1)*-th subproblem.

***Linking constraints***

*void SetLink (int NumLink, char * SenseLink, int * RhsLink, bool * PresentLink)*

Sets the number, sense, and right-hand side of linking constraints. The sense of each constraint may be *'L'*, *'E'* or *'G'*. The last argument is used to tell which linking constraints will be kept in all (restricted) master problems. If *PresentLink[i]==true* then the *(i+1)*-th linking constraint will be present in the first (restricted) master problem (RMP) and never deleted. If *PresentLink[i]==false* then the *(i+1)*-th linking will not be present in the first RMP and may be deleted. This last option can be overriden by the input parameter *ParDynLink*.

*void SetDEqual (bool DEqual)*

If *DEqual==true* then $D^1=...=D^h=D$.

If this function is not called it is assumed that all *D* matrices are different. After setting *DEqual* to true, function *SetD(...)* (and not *SetDk(...)*) must be called.

*void SetD (int *DBeg, int *DInd, double *DVal, int DNnz)*

Defines the matrix *D* in the case $D=D^1=...=D^h$. Must be called after a call to *SetDEqual(true)*.

*DBeg* is an array of length *NumVars* where *DBeg[j]* contains the first index in *DInd* and *DVal* related with the *(j+1)*-th variable. *DInd* and *DVal* must contain the indices and values, respectively, in the original rows (excluding constraints related with subproblem – original model –, or convexity constrains – master model). *DNnz* is the length of the arrays *DInd* and *DVal*.

*void SetDk (int ** DkBeg, int ** DkInd, double ** DkVal, int * DkNnz)*

Same as *SetD(...)* but for the case where the *D* matrices are not all equal. For each subproblem *k*, the meaning of the arguments is equivalent to the one explained in *SetD(...)*.

***Extra variables***

*void SetE (int * EBeg, int * EInd, double * EVal, int ENnz)*

Sets the *E* matrices. *EBeg* is an array of length *NumExtraVars* where *EBeg[j]* contains the first index in *EInd* and *EVal* related with the *(j+1)*th extra variable. *EInd* and *EVal* must contain the indices and values, respectively, in the rows of the master (including convexity constraints, which allows using artificial variables in those constraints). *ENnz* is the length of the arrays *EInd* and *EVal*.

*void SetObjExtraVars (double * ObjExtraVars)*

Sets the coefficients of the extra variables in the objective function. *ObjExtraVars[j]* contains the coefficient of the *(j+1)*-th extra variable.

*void SetArtCost (double aArtCost)*

Sets the artificial cost value. An artificial cost is needed for keeping feasibility of the master in all nodes of the tree, since a hidden artificial variable is used there.

***Convexity constraints***

*void SetConv (int NumConv, char * SenseConv = NULL, int * RhsConv = NULL,*
*bool * PresentConv = NULL)*

Sets the number, sense, and right-hand side of convexity constraints. The sense of each may be *'L'*, *'E'* or *'G'*. The last argument is used to define which convexity constraints will be kept in all (restricted) master problems. If *PresentConv[i]==true* then the *(i+1)*-th convexity constraint will be present in the first (restricted) master problem (RMP) and never deleted. If

*PresentConv[i]==false* then the *(i+1)*-th linking will not be present in the first RMP and may be deleted. This last option can be overriden by the input parameter *ParDynConv* (defined in the next section).

***Subproblem definition***

*void SetSubproblem (int * NumConstraintsk, char ** Sensek, double ** Rhsk, int ** AkBeg, int ** AkInd, double ** AkVal, int * AkNnz)*

Defines the default subproblem(s). That is, the yellow blocks in Figure 6.1, page 155 of the main text.

*void SetSubproblem (subproblem *MySubproblem)*

If a specific subproblem solver or specific branching rules are to be used, an object of a class derived from the base class *subproblem* is passed by calling this function. In this case, it is up to the *MySubproblem* object to know about the model he will be asked to solve.

## Parameters

In order to use *ADDing* a parameters input file must be given. Each line of that file may be:

- Empty.
- Start with * (comment line).
- Of the format \t* (comment line).
- Of the format *Value* \t*Parameter*.

The *Value* field gives the value for the parameter specified by the field *Parameter*. Any missing parameter will take its default value. An example of a parameter file is given in Figure A.1.

```
          ******************
          * General        *
          ******************

          ParMaxTime
false     ParOnlyRoot
          ParWriteSolution

          **************
          * TREE       *
          **************

          ParMaxOptimisedNodes
          ParTypeSearch
          ParUpBranchFirst
-1        ParTypeBranch
          ParRestartIterCount
          ParRootHeur

          **********************
          * FIRST RMP         *
          **********************
true      ParExtraSetFirstRMP
false     ParSPFirstRMP
10        ParHeurFirstRMP

          **********************
          * RMP OPTIMISATION  *
          **********************

          ParRMPSolver
20        ParIterExactRMP
.001      ParMaxIterRMP

          **********************
          * COLS MANAGEMENT   *
          **********************

2         ParColsRemove
```

```
10          ParIterColsRemove
            ParColsMaxInactivity

            ************************
            * CONV ROWS MANAGEMENT *
            ************************

false       ParDynConv
            ParConvRemove
            ParIterConvRemove
            ParConvMaxInactivity
            ParIterConvInsert

            ************************
            * LINK ROWS MANAGEMENT *
            ************************

true        ParDynLink
            ParLinkRemove
            ParIterLinkRemove
            ParLinkMaxInactivity
10000       ParIterLinkInsert

            *********************
            * SUBPROBLEM        *
            *********************

            ParSolveSPHeur
            ParExtraHeurSols
            ParIterExtraHeurSols
            ParNumSPSolsAsked

            *********************
            * LCI CUTS          *
            *********************

            ParTypeLCICuts
            ParDepthLCICuts
            ParFirstLCICuts
            ParFreqLCICuts


            **************
            * TOLERANCES *
            **************

            ParTolVar
            ParTolObj
```

Figure A.1 Example of a parameters file.

We now describe the meaning of each parameter.

***General***

*int ParMaxTime*

Maximum time, expressed in seconds, allowed to solve the problem.

A negative value means no time limit is set. Possible values: any positive integer. Default: 3600.

*bool ParOnlyRoot*

Specifies wether only the root is to be solved or not.

Possible values: “true”, only the root node is solved; “false”, the integer problem is solved. Default: “false”.

*bool ParWriteSolution*

Defines if the obtained solution should be written to a file.

Possible values: “true”, the obtained solution (in the original space) is written in a text file with the same name as the results file, with the extension 'sol'; “false”, the obtained solution is not written. Default: “false”.

*int ParMaxOptimisedNodes*

Limit to the number of optimised nodes of the search tree. A non-positive value means no limit is imposed. Default: 0.

***Tree***

*int ParTypeSearch*

Search tree strategy. Possible values: 1, depth first; 2, breadth first; 3, best first; 4, depth until an incumbent is found and then best; 5, depth when branching occurs, best in the other situations. Default: 5.

*int ParTypeBranch*

Branching rule. For every default branching rule, binary variables have priority over general integer ones. Specific branching rules can be implemented in the class *MySubproblem*. Those should be assigned with negative integer values. Possible values: 1, fractional variable with fractional part closest to 0.5; 2, first fractional variable found; 3, fractional variable with fractional part closest to 1; 4, fractional variable with fractional part closest to 0. Default: 1.

*bool ParUpBranchFirst*

Specifies if priority should be given to up branches (last branching constraint of type “≥”) or down branches (“≤”). Possible values: “true”, first up branch is chosen before the down branch; “false”, first down branch is chosen before the up branch. Default: “false”.

*bool ParRestartIterCount*

Defines how the iteration count is performed in the nodes of the tree, other than the root. If this parameter takes value “true” then the iteration counter of column generation is reset in every node of the tree. That means that, for example, if the columns removal is performed every 10th iteration and all the nodes of the tree are solved in less than 10 iterations then no columns removal is performed in the tree. The same applies to all parameters that depend on the iteration count. Possible values: “true” and “false”. Default: “false”.

*bool ParRootHeur*

Defines wether a RMP heuristic is used at the root or not. The RMP heuristic consists in solving with Cplex the MIP associated with the last RMP obtained. Currently, the time spent in Cplex is limited to 60 seconds (more elaborated alternatives will be explored in a near future). Possible values: “true” and “false”. Default: “false”.

***First restricted master problem (RMP)***

*bool ParExtraSetFirstRMP*

Controls if an extra set of extreme points and/or rays are asked to the subproblem in the construction of the first RMP. The availability of those additional columns are of the responsibility of the class *MySubproblem*. Possible values: “true” and “false”. Default: “false”.

*bool ParSPFirstRMP*

Controls if the first RMP includes columns associated with the optimal solutions of the subproblems solved with the original costs. Possible values: “true” and “false”. Default: “false”.

*char* ParHeurFirstRMP*

Heuristics must be implemented in the *MySubproblem* class. This parameter specifies which ones will be used in constructing the first RMP. Possible values: "-", do not use heuristics; any binary string, for instance “01010” means that the second and fourth heuristics will be used in constructing the first RMP (in this example, if *MySubproblem* only provides e. g. two heuristics, of course the fourth will not be used). Default: "1".

***RMP optimisation***

*char* ParRMPSolver*

Solver of the RMPs. Possible values: “Cplex_P”, Cplex primal; “Cplex_D”, Cplex dual; “Cplex_N”, Cplex network followed by dual; “Cplex_B”, Cplex barrier. Default: “Cplex_D”.

*int ParIterExactRMP*

*ADDing* implements a non-standard procedure to deal with decompositions where the RMPs are particularly difficult to solve. The procedure consists in not solving the RMP exactly, but only performing a given number of simplex iterations (or finding a feasible dual solution if that number of iterations was not sufficient to find one), in some column generation iterations. Currently, ParIterExactRMP gives the initial frequency for solving exactly the RMP. Every 10 iterations that frequency is decreased by one, until the RMP is solved exactly in all iterations. This strategy is particularly devised for generating quickly a large number of columns in the first iterations in order to obtain feasible solutions. Other strategies will be explored in a near future. This parameter will have effect only if the parameter *ParRMPSolver* is set to *“Cplex_D”*. Possible values: any positive integer. Default: 1.

*double ParMaxIterRMP*

Sets the maximum number of (dual) simplex iterations when solving the RMP inexactly. If a feasible dual solution is not found in that number of iterations, the optimisation proceeds until that occurs. This parameter is irrelevant if *ParExactRMP* is set to 1 or *ParRMPSolver* is not "*Cplex_D*". The unit measure of this parameter is equal to the number of convexity constraints plus the number of linking constraints. Possible values: any positive fractional. Default: 0.5.

***Columns management***

*int ParColsRemove*

Parameter that controls the removal of columns. Possible values: 0, no columns removal is performed; 1, remove all columns inactive for more than *ParColsMaxInactivity* (see below) iterations; 2, remove columns if their number exceed three times the number of linking constraints plus the number of convexity constraints (starting with the ones with larger reduced cost); 3, remove all columns with reduced cost greater than the gap. Default: 0.

*int ParIterColsRemove*

Parameter that controls the frequency of columns removal (see *ParColsRemove*). Possible values: any positive integer. Default: 5.

*int ParColsMaxInactivity*

A column is inactive in one iteration if its reduced cost is greater than 1. If that happens for more than *ParColsMaxInactivity* consecutive iterations, that column is removed (if *ParColsRemove* is set to 2*)*. Possible values: any positive integer. Default: 10.

***Convexity rows management***

*bool ParDynConv*

If this parameter is set to "true", convexity rows can be managed dynamically (that is, inserted and/or removed in some iterations) according to other parameters (see below). In that case, it is up to the decomposition model to specify which convexity rows cannot be removed. Setting this parameter to "false" overrides any intention of the decomposition model. Possible values: "true" and "false". Default: "false".

*int ParConvRemove*

Parameter that controls the removal of convexity rows. It only has effect if *ParDynConv* is set to "true". Possible values: 0, no convexity rows removal is performed; 1, all rows that were inactive for the last *ParConvMaxInactivity* (see below) iterations. Possible values: any non-negative integer. Default: 0.

*int ParIterConvRemove*

Parameter that controls the frequency of convexity rows removal. Only has effect if *ParDynConv* is set to "true" and *ParConvRemove* > *0*. Possible values: any nonnegative integer. Default: 5.

*int ParConvMaxInactivity*

A row is inactive if it has slack or is of the form "0=0". If that happens for more than *ParConvMaxInactivity* consecutive iterations, that row is removed (if *ParConvRemove* is set to 1*)*. Possible values: any positive integer. Default: 10.

*int ParIterConvInsert*

Frequency of violated convexity rows test. Only has effect if *ParDynConv* is set to "true". Possible values: any positive integer. Default: 1.

***Linking rows management***

*bool ParDynLink*

Same as *ParDynConv* (see above) but for linking constraints. Possible values: "true" and "false". Default: "false".

*int ParLinkRemove*

Same as *ParConvRemove* (see above) but for linking constraints. Possible values: 0, no linking rows removal is performed; 1, all linking rows that were inactive for the last *ParLinkMaxInactivity* (see below) iterations. Possible values: any nonnegative integer. Default: 0.

*int ParIterLinkRemove*

Same as *ParIterConvRemove* (see above) but for linking constraints. Possible values: any positive integer. Default: 5.

*int ParLinkMaxInactivity*

Same as *ParConvMaxInactivity* (see above) but for linking constraints. Possible values: any positive integer. Default: 10.

*int ParIterLinkInsert*

Same as *ParIterConvInsert* (see above) but for linking constraints. Possible values: any positive integer. Default: 1000.

***Subproblem***

*int ParSolveSPHeur*

The subproblem can be solved heuristically in all iterations. Whenever no attractive columns are generated, then an exact solver is used to check for optimality. This parameter tells which heuristic should be used in order to solve the subproblem heuristically. Possible values:

−1, do not solve the subproblem heuristically; n, use the n-th heuristic. Default: −1.

*char* ParExtraHeurSols*

In some iterations (see below, *ParITerExtraHeurSols*), heuristics for the subproblem can be used to obtain extra columns. This parameter defines which heuristics should be used. Possible values: "-", do not use heuristics; any binary string (for example, “01010” means that the second and fourth heuristics will be used to heuristically generate extra columns). Default: "01".

*int ParIterExtraHeurSols*

Frequency of extra columns generation through heuristics. It may be irrelevant depending on *ParExtraHeurSols*. Possible values: any integer (a negative value means extra heuristic columns will not be generated). Default: −1.

*int ParNumSPSolsAsked*

The *subproblem* class (in fact, currently only a derived class, *MySubproblem*) may be able to generate the second best, the third best, and so on, subproblem optimal solutions. This parameter controls the number of times the subproblem is asked to provide those sub-optimal solutions, as long as their columns are attractive. Possible values: any integer ≥0. Default: 0.

***Lifted cover inequalities (LCIs)***

The next parameters should not be used. The inclusion of LCIs cuts is still at an experimental phase.

*int ParTypeLCICuts*

Type of LCI cuts to be used. Possible values: −1, do not use LCIs; 0, generate general LCIs as described in Gu, Nemhauser and Savelsbergh (1998); generate simple LCIs as described in Gu, Nemhauser and Savelsbergh (1998). Default: −1.

The LCI cuts will be used in the nodes of the search tree according to the next three parameters, which are cummulative.

*int ParDepthLCICuts*

Possible values: −1, do not use this parameter; 0, only in the root; n (positive integer): in nodes with depth ≤n. Default: −1.

*int ParFirstLCICuts*

Possible values: −1, do not use this parameter; n (positive integer): first n optimised nodes. Default: −1.

*int ParFreqLCICuts*

Possible values: −1, do not use this parameter; n (positive integer): every n-th optimised

node. Default: −1.

*double ParCutTail*

If the percentage improvement is not higher than this parameter, no more cuts are generated in the iteration. Possible values: any nonnegative fractional. Default: 0.001.

***Tolerances***

*double ParTolVar*

Used for checking infeasibily (if the total value of artificial variables in an optimal RMP exceeds *ParTolVar*, then the node is unfeasible) and the integrality of the variables (a variable with a value with a fractional part smaller than *ParTolVar* or larger than *1–ParTolVar* is considered as integer). Default: 1e−5.

*double ParTolObj*

Tolerance for the attractiveness of a column. Default: 1e−5.

## Results

The output of *ADDing* includes a file with results. In Figure A.2 an example of such a file is given.

```
            Model info
320         NumSPs
192         NumVars
192         NumArts
0           NumExtraVars
0           NumLinVars
192         NumBinVars
0           NumIntVars
320         NumConv
6144        NumOrig

            General
2797.75     TimeBPC
2797.531    TimeBPCOpt
0           StatusBPC

            Tree
0.079       TimeTree
15          NumOpt
3           NumPrunedAfterOpt
3           NumIncumbentUpdates
4           NumPrunedWithoutOpt
9           NumParents
0           NumPrunedDuringOpt
0           NumUnfeasible
4           NodeFirstIncumbent
3134980     FirstIncumbentValue
6           MaxQueueSize
3132695     ValueIncumbent

            Column generation
0.156       TimeCGRMPFirst
2229.047    TimeCGOptRoot
2025.23     TimeCGRMPOptRoot
40.364      TimeCGSPOptRoot
567.656     TimeCGOptNode
461.212     TimeCGRMPOptNode
28.371      TimeCGSPOptNode
2527        NumIterRoot
1227        NumIterNode
808640      NumCGSPsSolvedRoot
27175       NumCGSPsPointRayRoot
392640      NumCGSPsSolvedNode
3135        NumCGSPsPointRayNode
81631       NumCGRMPLargestColsRoot
3286        NumCGRMPLargestRowsRoot
84766       NumCGRMPLargestColsNode
3437        NumCGRMPLargestRowsNode
0           NumExtremeRays
619325      ValueCGRMPFirst

            Node solver
0           LCICutGenRoot
0           LCICutGenNode
0           NumIterCutRoot
0           NumIterCutNode
0           NumLCICutRoot
0           NumLCICutNode
3132002.5   ValueRoot
```

Figure A.2 Example of a results file.

## Deriving the *MySubproblem* class

***Member data to be used by the derived class***

*double** ModCosts*

Double array of modified costs: *ModCosts[k][j]* contains the objective function coefficient of the (*j+1*)-th variable of the (*k+1*)-th subproblem. The user does not need to worry about memory allocations and deallocations.

*int NumHeurs*

In the constructor of *MySubproblem*, *NumHeurs* should be set to the number of heuristics the class implements.

*bool *IsAggregated*

In the constructor of *MySubproblem*, this array should be allocated with size *NumHeurs*. The entry with index *j* should be *true* if the *(j+1)*-th heuristic is an aggregated one, and *false*, otherwise. In an aggregated heuristic the solution of one subproblem influences the others. In an disaggregated heuristic each subproblem is solved independently. We note that the extreme points will always be inserted in the RMP in a disaggregated manner. Currently *ADDing* does not support aggregated columns (that is, one column being related with more than one subproblem).

*ParTypeBranch*

This copy of the input parameter allows the user to define more than one branching rule.

***Member virtual functions related with implementing a specific subproblem solver***

Only the two first member functions (*SetSP(...)* and *Optimise(...)*) must be implemented in order to have a specific subproblem solver. All the others are optional.

*virtual void SetSP (int k, double *Costs)*

Prepare the (*k+1*)-th subproblem to be solved by setting the modified costs of all its variables. Those values should be written in *ModCosts*.

*virtual int Optimise(int k, pointray *PointRay, double &Value)*

Optimises the (*k+1*)-th subproblem.

Return values: 0, no relevant solution was found; 1, the optimal extreme point, or an extreme ray, was found, has value *Value* and is given in *PointRay*; 2, no feasible solution was

found.

All the user must set inside this function related with the class *pointray* are the following two member functions:

*void pointray::SetType(int Type)*

*Type* should be 1 if it is an extreme point and 2 if it is an extreme ray;

*void pointray::AddVar(int IdxSP, int IdxVar, double ValVar)*

Adds a variable of the subproblem *IdxSP* with index *IdxVar* and value *ValVar* to the description of the extreme point or extreme ray.

*virtual int OptimiseNext(int k, pointray *PointRay, double &Value)*

This function is similar to *Optimise(...)*. Its purpose is to return the second best solution, third best solution, and so on. If the input parameter *ParNumSPSolsAsked* is greater than zero, this function will be called after a call to *Optimise(...)* in order to include additional (sub)optimal extreme points (or extreme rays) in the RMP.

*virtual int SolveHeurDis (int k, pointray *PointRay, double &Value, int Heuristic)*

This member function is meant to solve the subproblem *k* with the *(Heuristic+1)*-th heuristic (which is a disaggregated one; for the difference regarding an aggregated one, see the member data *IsAggregated*, above). The meaning of the other arguments is the same as in *Optimise(...)*.

Return values: 0, no extreme point or ray was found; 1, one extreme point or ray was found.

*virtual int SolveHeurAgg (pointray **PointRay, double *Values, int Heuristic)*

Similar to *SolveHeurDis(...)* but for an aggregated heuristic: a set of extreme or rays may be obtained.

Return value: number of extreme points and rays found. The double array *PointRay* should have one extreme point or ray in each entry.

*virtual int GetUpperNumExtreme ()*

Returns an upper bound to the number of additional extreme points to be inserted in the first RMP.

*virtual int GetSetExtreme (pointray **PointRay)*

The argument should be fed with a set of points/rays to be inserted in the first RMP (an upper bound to their number given by the return value of *GetUpperNumExtreme()*). The return value is the effective number of extreme points and rays generated by the function. This implementation of *GetUpperNumExtreme()* and *GetSetExtreme(...)* avoids the need for memory allocations and deallocations by the user.

***Member virtual functions related with specific branching rules***

*virtual int GetNumBranches()*

Returns the number of branches that *MySubproblem::GetBranches(...)* generates. The base class implementation returns the value 2.

*virtual void GetBranches(constraint **BranchConstraint)*

This member function creates the branching constraints. Using *ParTypeBranch*, more than one branching rule can be implemented in it.

The argument is an array of objects (in fact, pointers to objects) of class *Constraint* filled with the constraints. The relevant member functions of *constraint* are

void *SetSense (char Sense);*

void *SetRhs (int Rhs);*

*void AddVar(int IdxSP, int IdxVar, double Coef).*

As an example, if two branching constraints are created based on a single binary the instructions given in Figure A.3 should be included, where *BranchSP* is the index of the subproblem of the variable given by index *BranchVar* and that variable has coefficient *1* in both constraints. Branching on several variables can be done by several calls to *AddVar(...)*.

```
...
rBranchConstraint[0]->AddVar(BranchSP,BranchVar,1);
rBranchConstraint[0]->SetSense('L');
rBranchConstraint[0]->SetRhs(0);
rBranchConstraint[1]->AddVar(BranchSP,BranchVar,1);
rBranchConstraint[1]->SetSense('G');
rBranchConstraint[1]->SetRhs(1);
...
```

Figure A.3 Example of instructions to create two branching constraints.

In order to specify the branching rule, the *subproblem* class has a pointer to an object representing the current (fractional) solution. The only member function of that class the user must be aware of is *double originalsolution::GetVarValue(int k, int j)*, which returns the value of the variable indexed by *j* of the subproblem indexed by *k*. The base class already has a pointer to the object where the current solution is kept, with name *OriginalSolution*.