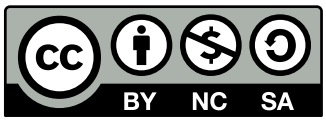
Aalto University
School of Science

Master's Programme in Mathematics and Operations Research

A branch-and-price algorithm for the minimum flow decomposition problem

Tuomas Porkamaa

Master's Thesis
2026



Aalto University
School of Science

Author Tuomas Porkamaa

Title A branch-and-price algorithm for the minimum flow decomposition problem

Degree programme Mathematics and Operations Research

Major Applied Mathematics

Supervisor Asst. Prof. Philine Schiewe

Advisor Dr. Fernando Dias

Date 31 July 2026 **Number of pages** 49 **Language** English

Abstract

The minimum flow decomposition problem is a common problem in computer science and combinatorial optimization. Its objective is to decompose a network flow into a minimum-cardinality set of weighted source-sink paths whose aggregate flow exactly reproduces the original network flow. Because the problem is NP-hard, most proposed algorithms do not solve it exactly, but instead rely on heuristics or consider simpler variants that are solvable in polynomial time. Exact solution methods have been proposed only recently. These methods formulate the problem as an integer linear program and use optimization solvers to obtain an optimal solution.

A branch-and-price algorithm combines branch-and-bound with column generation to solve large-scale integer linear programs. It is often used as a standard solution method for formulations that provide stronger LP relaxations at the expense of increased number of variables.

In this thesis, we propose a branch-and-price algorithm as an exact solution method for the minimum flow decomposition problem. We derive an alternative stronger ILP formulation, whose LP relaxation is dynamically solved by a shortest-path based column generation algorithm. We also propose two competing branching strategies that do not complicate the column generation phase. Our results show that the algorithm involving a supplementary branching step solves instances faster compared to the conventional branch-and-bound approach.

Keywords Combinatorial optimization, integer linear programming, minimum flow decomposition, branch-and-price, column generation, Dantzig–Wolfe decomposition, directed acyclic graph

Aalto University
School of Science

Tekijä Tuomas Porkamaa

Työn nimi Branch-and-price algoritmi pienimmän virtaushajotelman ongelmalle

Koulutusohjelma Matematiikka ja operaatiotutkimus

Pääaine Sovellettu matematiikka

Työn valvoja Apulaisprofessori Philine Schiewe

Työn ohjaaja Tohtori Fernando Dias

Päivämäärä 31.7.2026 **Sivumäärä** 49 **Kieli** englanti

Tiivistelmä

Pienimmän virtaushajotelman ongelma on yleinen ongelma tietojenkäsittelytieteessä ja kombinatorisessa optimoinnissa. Sen tavoitteena on hajottaa verkon virtaus mahdollisimman pieneksi joukoksi painotettuja polkuja alku- ja loppusolmun välillä siten, että polkujen yhteenlaskettu virtaus vastaa täsmälleen verkon alkuperäistä virtausta. Koska ongelma on NP-vaikea, useimmat ehdotetut algoritmit eivät ratkaise sitä optimaalisesti, vaan turvautuvat heuristisiin menetelmiin tai käsittelevät ongelman yksinkertaisempia, polynomisessa ajassa ratkaistavia muunnelmia. Tarkkoja ratkaisumenetelmiä on esitetty vasta viime vuosina. Nämä menetelmät muotoilevat ongelman lineaariseksi kokonaislukuoptimointitehtäväksi ja hyödyntävät ratkaisuohjelmistoja optimaalisen ratkaisun löytämiseksi.

Branch-and-price-algoritmi eli haarauttamis- ja hinnoittelualgoritmi yhdistää branch-and-bound-menetelmän eli haarauttamis- ja rajoitusmenetelmän sekä sarakegenerointimenetelmän ratkaistakseen lineaarisia kokonaislukuoptimointitehtäviä, jotka sisältävät suuren määrän muuttujia. Sitä käytetään usein vakiintuneena menetelmänä optimointimalleille, jotka tuottavat vahvemman LP-relaksaation suuremman muuttujamäärän kustannuksella.

Tässä opinnäytetyössä esitimme branch-and-price-algoritmin pienimmän virtaushajotelman ongelman täsmälliseksi ratkaisumenetelmäksi. Johdimme ensin vaihtoehtoisen vahvemman optimointimallin, jonka LP-relaksaatio ratkaistiin dynaamisesti lyhyimmän polun ongelmaan perustuvalla sarakegenerointialgoritmilla. Kokonaislukuarvoisten ratkaisujen saavuttamiseksi, esitimme kaksi vaihtoehtoista haarauttamisstrategiaa, jotka eivät monimutkaista sarakegeneroinnin toimintaa. Tuloksemme osoittivat, että täydentävän haarauttamisvaiheen sisältävä branch-and-price algoritmi ratkaisi testitapaukset nopeammin kuin tavanomainen branch-and-bound menetelmä.

Avainsanat Kombinatorinen optimointi, lineaarinen kokonaislukuoptimointi, minimivirtaushajotelma, branch-and-price, sarakegenerointi, Dantzig-Wolfe hajotelma, suunnattu syklitön graafi

Contents

Symbols and abbreviations

Symbols

$G = (V, E, f)$	Flow network
V	Set of nodes
E	Set of arcs
$\mathbb{N}$	Set of nonnegative integers
s, t	Source and sink nodes
f_e	Flow of an arc e
P	Set of all s–t paths
f_p	Bottleneck flow of path p
H	Upper bound on the minimum flow decomposition size
δ_{ep}	Indicator that arc e belongs to path p
$1(\cdot)$	Indicator function
y_h, x_e^h, w_h, z_e^h	Compact formulation variables
$\lambda_{hp}^0, \lambda_{hp}^1$	Extensive formulation variables

Abbreviations

DAG	Directed acyclic graph
DW	Dantzig–Wolfe
LP	Linear program
ILP	Integer linear program
MFD	Minimum flow decomposition
RMP	Restricted master problem
PP	Pricing problem

1 Introduction

This thesis studies an exact solution method for the Minimum Flow Decomposition (MFD) problem, which is a common problem in computer science and combinatorial optimization and widely applied in the field of bioinformatics, network routing, and transportation planning. The objective of the MFD problem is to decompose a network flow into a minimum-cardinality set of source-sink or $s-t$ paths, each one carrying a weight, such that the aggregated path flows perfectly explain the flow values on the edges of the network.

The MFD problem was first formally defined by Vatinlen et al. [49], motivated by problems in telecommunication network engineering. They proved the problem NP-hard and derived the properties of optimal solutions, such as the linear independence of selected paths. They also developed two heuristic methods, *shortest path heuristic* and *maximal residual path capacity heuristic*, which construct a decomposition by iteratively extracting saturating paths from the residual graph. Finally, they proposed simple upper and lower bounds for the ground truth decomposition size.

Hartman et al. [22] proved that the MFD problem remains NP-hard even if the arcs have only three distinct flow values. They also showed that it is NP-hard to find a decomposition whose number of paths is within a factor of at most $\beta > 1$ of the optimal number Z^*, and that the greedy algorithms proposed by Vatinlen et al. [49] can be provably far from optimal. In the worst case, these algorithms may produce approximate decompositions whose size is $\Omega(\sqrt{|E|}\, Z^*)$, where $|E|$ is the number of arcs in the network. A new two-criteria approximation method was proposed, capable of decomposing at least $(1-\epsilon)$ proportion of the flow using $O(Z^*/\varepsilon^2)$ paths.

Mumey et al. [31] proposed two polynomial-time algorithms, *OddCover* and *JointOddCover*. These algorithms focus on finding flow decompositions where the path flows are powers of two. They produced solutions using $O(Z^*\, \Delta^{\log K} \log K)$ paths, where Δ is the maximum length of a $s-t$ path and K is the largest arc weight value in the network. Motivated by transcript assembly and multiple-genome assembly problems, Shao and Kingsford [38] proposed an efficient pseudopolynomial-time heuristic called *Catfish*. Their heuristic was developed by identifying that the gap between the optimal decomposition size and a known upper bound is determined by the nontrivial equations within the arc flow values.

In addition to approximation algorithms, exact solution methods have also been recently proposed. The first practical and exact MFD solver was proposed by Dias et al. [14] who modeled the problem as an integer linear program (ILP). Their study was motivated by the need for exact solutions in RNA assembly, where ground-truth transcripts and their abundances should correspond to the minimum flow decomposition solution. While exact solvers for multiassembly problems have already been proposed [28], these methods require explicitly enumerating all possible $s-t$ paths and assigning a binary variable to each path to indicate whether it is selected in the optimal solution. Because the number of paths scales exponentially with the size of the network, these methods are only practical for moderately small network instances. On the other hand, the model proposed by Dias et al. constructs each path using arc-level variables.

In their experiments, Dias et al. compared their exact ILP approach with the approximation heuristics Catfish [38] and Toboggan [28]. Their method found optimal flow decomposition solutions for RNA assembly networks in under 13 seconds for every instance. This was over three times faster than Catfish, while Toboggan took over 60 seconds in some instances. ILP models for two MFD problem variants were also proposed. The first variant finds solutions satisfying a specific set of subpath constraints, while the second variant is used in situations where arc weights are assumed to lie within intervals rather than being fixed integers.

On the application side, flow decompositions and their variants are widely used in computational biology, especially for reconstructing biological sequences from RNA sequencing data [6, 16, 27, 42, 51] or viral quasispecies [2]. Many RNA sequencing pipelines construct a weighted directed acyclic network (DAG) whose vertices represent exons, and arcs and their weights represent splice junctions and their observation counts, respectively, between exons. A source-to-sink path in this network represents one reconstructed RNA transcript, and the assigned weight represents its expression level [51]. The minimum-sized decompositions are then often preferred, as they usually best explain the observed data [16].

In network routing [7, 22, 24, 31], the flow network often represents network devices or their locations and their connections via bandwidth-limited communication links. The explicit transmission routes are then modeled as paths in the network, where the corresponding weight represents the amount of network traffic on the route. Finding a minimum flow decomposition over this network is important because each path consumes entries in the tables of the routers that it goes through, and a smaller number of paths makes the network configuration more efficient [22, 31].

In transportation planning problems, such as evacuation routing [32] or public transport scheduling [33], a flow network usually represents locations and their connections, or location-time events, such as arrival or departure times and waiting periods. A weighted path on this network represents a complete route or a vehicle schedule, where the weight equals to the number of vehicles serving it. Flow decompositions with a small number of paths are preferred because they reduce the number of routes or vehicle rotations that need to be operated and managed.

2 Preliminarities

This section introduces the main definitions and optimization methods that are used throughout the thesis. We begin by defining flow networks and k-flow decompositions, after which we present two integer linear programming formulations for the minimum flow decomposition problem. We then discuss the formulation strength and LP-equivalence, which provide the basis for comparing different ILP formulations. After that, we explain the column generation procedure for solving large linear programs efficiently. Finally, we present the Dantzig-Wolfe decomposition method and branch-and-price framework.

2.1 Flow decomposition

A directed graph is defined as a pair $G = (V, E)$ where V and $E \subseteq V \times V$ are the sets of nodes/vertices and arcs/edges, respectively. A directed path from node u to node v is a sequence of nodes $(u = v_0, \ldots, v_n = v)$ such that $(v_{i-1}, v_i) \in E$ for all $i = 1, \ldots, n$ and no node is repeated. Equivalently, one may define a directed path as a sequence of arcs $((v_{i-1}, v_i))_{i=1}^{n}$ instead. A directed cycle is a directed path $(v_0, ..., v_n)$ together with the arc (v_n, v_0). A node $s \in V$ is called a *source* if it has no in-coming arcs, and analogously, a node $t \in V$ is called a *sink* if it has no out-going arcs [14]. From now on, we assume that G is a *directed acyclic graph* (DAG) meaning it does not contain directed cycles. [1]

Definition 1 (Flow network [14])**.**
Let (V, E) be a DAG with an unique source and sink nodes s and t, respectively, and let $f : E \to \mathbb{Z}_{>0}$ define a positive integer weight f_{uv} for every arc $(u, v) \in E$. Then a tuple $G = (V, E, f)$ is said to be a flow network if it satisfies the following flow conservation property

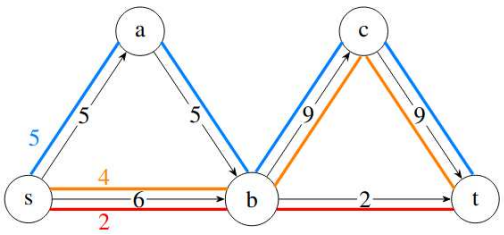
$$\forall\, v \in V \backslash \{s, t\} : \quad flow_{in}(v) = \sum_{(u,v)\in E} f_{uv} = \sum_{(v,w)\in E} f_{vw} = flow_{out}(v). \tag{1}$$

The total flow of G is defined to be either the out-flow from source or in-flow to sink

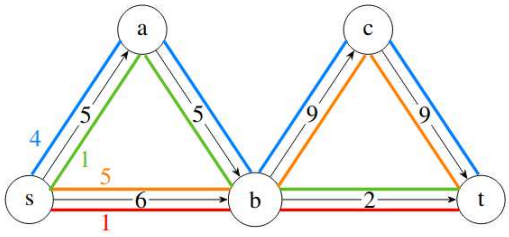
$$flow(G) = \sum_{(s,u)\in E} f_{su} = \sum_{(u,t)\in E} f_{ut}. \tag{2}$$

Definition 2 (k-Flow decomposition [14])**.**
Given a flow network $G = (V, E, f)$, a k-flow decomposition is a pair $(\mathcal{P}, w)$ where $\mathcal{P} = (p_1, ..., p_k)$ is a set of source-to-sink ($s - t$) paths associated with a set of path flows $w = (w_1, ..., w_k)$, where each $w_i \in \mathbb{Z}_{>0}$, such that the flow value of each arc equals the sum of the path flows of the paths using that arc

$$\forall e \in E : \sum_{\substack{i=1,\ldots,k: \\ e\in p_i}} w_i = f_e. \tag{3}$$



(a) A 3-flow decomposition with path weights (5, 4, 2).



(b) A 4-flow decomposition with path weights (4, 1, 5, 1).

Figure 1: Example of two flow decompositions.

Two example k-flow decompositions are illustrated in Figure 1. The minimum flow decomposition problem seeks to find a flow decomposition $(\mathcal{P}, w)$ such that $|\mathcal{P}|$ is minimized [14, 49]. Two ILP formulations for this problem are given below, a constructive compact arc-node formulation inspired by [14], and an enumerative extensive arc-path formulation.

Definition 3 (Compact arc-node formulation [14])**.**
Let $G = (V, E, f)$ be a flow network, and let H be the number of paths in some known feasible flow decomposition of G. Thus, H is an upper bound on the minimum flow decomposition size. For each $v \in V$, let $\text{in}(v)$ and $\text{out}(v)$ denote the sets of arcs entering and leaving v, respectively. Then a compact arc-node ILP formulation for MFD problem, later referred to as compact formulation, is given as

$$\min \quad \sum_{h=1}^{H} y_h \tag{4.1}$$

$$s.t. \quad \sum_{h=1}^{H} z_e^h = f_e \qquad \forall e \in E \tag{4.2}$$

$$\sum_{e \in in(v)} z_e^h - \sum_{e \in out(v)} z_e^h = \begin{cases} -w_h & \text{if } v = s \\ w_h & \text{if } v = t \\ 0 & \text{otherwise} \end{cases} \qquad \forall h \in \{1, \ldots, H\}, \forall v \in V \tag{4.3}$$

$$z_e^h \leq f_e x_e^h \qquad \forall h \in \{1, \ldots, H\}, \forall e \in E \tag{4.4}$$

$$y_h \leq w_h \qquad \forall h \in \{1, \ldots, H\} \tag{4.5}$$

$$\sum_{e \in in(v)} x_e^h - \sum_{e \in out(v)} x_e^h = \begin{cases} -y_h & \text{if } v = s \\ y_h & \text{if } v = t \\ 0 & \text{otherwise} \end{cases} \qquad \forall h \in \{1, \ldots, H\}, \forall v \in V \tag{4.6}$$

$$y_h \in \{0, 1\} \qquad \forall h \in \{1, \ldots, H\} \tag{4.7}$$

$$x^h \in \{0, 1\}^{|E|} \qquad \forall h \in \{1, \ldots, H\} \tag{4.8}$$

$$w_h \in \mathbb{N} \qquad \forall h \in \{1, \ldots, H\} \tag{4.9}$$

$$z^h \in \mathbb{N}^{|E|} \qquad \forall h \in \{1, \ldots, H\} \tag{4.10}$$

Every nonzero tuple (y_h, x^h, w_h, z^h) represents an element in $(\mathcal{P}, w)$. Variable x^h is the arc-incidence vector of a $s-t$ path, and y_h indicates whether this path is part of the decomposition. Variable w_h is the corresponding path-flow variable, and z^h is the arc-level representation of this flow. Constraints (4.2) ensure that (3) is satisfied, and constraints (4.6) ensure that the arc-incidence vector is a valid $s-t$ path, namely, if $y_h = 1$, there is a single arc leaving the source and arriving into the sink, and any other node has a single incoming and outgoing arc. If $y_h = 0$, then x^h is a zero vector. Constraints (4.3) ensure that a similar property holds for the arc-incidence flow vector z^h, i.e. it equals to $w_h x^h$ or a zero vector. Constraint (4.4) links the arc-flow vector z^h to the selected path x^h, forcing $z^h_e = 0$ whenever arc e is not used by path x^h_e. It also bounds the path flow w_h by the minimum arc-flow on the selected path. Constraint (4.5) cuts off degenerate solutions in which slot h is active, $y_h = 1$, but carries zero flow, $w_h = 0$. This constraint is not required for correctness, since any solution containing an active zero-flow slot can be replaced by one with $y_h = 0$ without increasing the objective value. Nevertheless, it is included because it strengthens the formulation (see Definition 7). Lastly, because G is assumed to be a DAG, we do not need separate constraints to ensure that each path is elementary. The objective function is to minimize the number of paths in a decomposition.

Definition 4 (Extensive arc-path formulation)**.**
Let $G = (V, E, f)$ be a flow network and P be the set of all $s-t$ paths in G. For each path $p \in P$, its bottleneck flow upper bound is given by $f_p = \min_{e \in p} f_e$. Then the extensive arc-path ILP formulation for MFD problem is given by

$$\min \quad \sum_{p \in P} y_p \tag{5.1}$$

$$s.t. \quad \sum_{p \in P} \delta_{ep} w_p = f_e \qquad \forall e \in E \tag{5.2}$$

$$w_p \le f_p y_p \qquad \forall p \in P \tag{5.3}$$

$$w_p \ge y_p \qquad \forall p \in P \tag{5.4}$$

$$w_p \in \mathbb{N} \qquad \forall p \in P \tag{5.5}$$

$$y_p \in \{0, 1\} \qquad \forall p \in P \tag{5.6}$$

Variable y_p denotes whether path p is part of the decomposition and w_p is the corresponding nonnegative integer-valued flow. Constraints (5.2) ensure that (3) is satisfied, where $\delta_p = \{\delta_{ep}\}_{e \in E}$ is the arc-incidence vector of path p. Constraints (5.3) ensure that paths not part of the decomposition, i.e., inactive paths, have flow values of 0, and the flows for active paths are bounded by f_p. Constraints (5.4) strengthen the formulation. The objective is to minimize the number of paths in a decomposition.

There are two main differences between the compact and extensive formulations given above, namely, their size, and the strength of the dual bounds they provide. Assuming that H is within a moderate range, the compact formulation has a smaller number of constraints and variables than the extensive formulation, which relies on the enumeration of all $|P|$ $s-t$ paths in G which is exponential to the number of vertices in G, namely $|P| \le 2^{|V|-1}$. We show later in the Section 2.3 that the extensive

formulation is stronger than the compact formulation, where we use the following common Definitions [3].

Definition 5 (Formulation)**.**
A polyhedral set $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ *is called a formulation for a set* $X \subseteq \mathbb{Z}^n$ *if* $X = P \cap \mathbb{Z}^n$.

Instead of only considering formulations that have the same dimension as X, it is often useful to have alternative formulations living in the higher dimensional space. For this purpose, we define the extended formulation as follows.

Definition 6 (Extended formulation)**.**
A polyhedral set $P = \{(x, y) \in \mathbb{R}^{n+p} : Ax + By \leq b\}$ *is called an extended formulation for a set* $X \subseteq \mathbb{Z}^n$ *if* $X = Proj_x(P) \cap \mathbb{Z}^n$ *where*

$$Proj_x(P) = \{x \in \mathbb{R}^n : \exists y \in \mathbb{R}^p \text{ s.t. } (x, y) \in P\}.$$

Definition 7 (Strong formulation and LP equivalence)**.**
Given a set $X \subseteq \mathbb{Z}^n$*, consider two extended formulations*

$$P_1 = \{(x, y) \in \mathbb{R}^{n+p_1} : Ax + By \leq b\}$$
$$P_2 = \{(x, y) \in \mathbb{R}^{n+p_2} : Cx + Dy \leq d\}$$

with corresponding ILPs

$$ILP_1 = \min\{c^T x_1 \quad s.t. \quad x_1 \in Proj_x(P_1) \cap \mathbb{Z}^n\}$$
$$ILP_2 = \min\{c^T x_2 \quad s.t. \quad x_2 \in Proj_x(P_2) \cap \mathbb{Z}^n\}$$

having LP relaxations LP_1 *and* LP_2*, respectively, obtained by dropping the integrality restrictions on the projected variables. Then we say that formulation* P_1 *is at least as strong as* P_2*, or* ILP_1 *dominates* ILP_2*, if* $Proj_x(P_1) \subseteq Proj_x(P_2)$*. In this case,* $Z_{LP_1} \geq Z_{LP_2}$ *holds for optimal LP solution values, meaning that* LP_1 *yields a tighter dual bound. Alternatively, if* $Proj_x(P_1) = Proj_x(P_2)$ *we say that the formulations are LP-equivalent.*

The notion of formulation strength and dual bound tightness is important because most ILP solvers are based on branch-and-bound and, in practice, branch-and-cut, where they repeatedly solve LP relaxations and then enforce integrality by branching and/or adding cutting planes [21, 25]. In a minimization setting, every node v of the branch-and-bound search tree corresponds to a linear problem whose optimal solution value is a lower (i.e., dual) bound on the objective value of every integer-feasible solution in the corresponding subtree originating from v. If this dual bound is greater than or equal to the objective value of the best incumbent integer solution so far, then the subtree cannot contain an improving solution, and the node can be pruned. If two formulations P_1 and P_2 share the same set of integer-feasible solutions (up to a projection), but P_1 is stronger than P_2, then, under the same branching decisions, the formulation P_1 yields a lower bound that is at least as large as that induced by P_2. This typically allows the solver to prune more nodes earlier and to explore a smaller search tree.

2.2 Reduced cost and column generation

In this section, we introduce the concept of reduced cost, which is essential for investigating the optimality of a solution to a linear programming problem. We also present an implicit enumeration method called column generation for identifying variables with negative reduced costs.

Consider the following linear programming problem in a standard form

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \quad (\alpha) \\ & x \geq 0 \end{aligned} \tag{6}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $m < n$ and α denotes the dual variables associated with the constraints. The coefficient matrix is represented as a sequence of columns, $A = (A^1, ..., A^n)$ for which we associate a nonsingular submatrix called *a basis* $A_{\mathcal{B}} = (A^j)_{j \in \mathcal{B}}$, $\mathcal{B} \subseteq \{1, ..., n\} = J$. The columns not part of the basis are denoted by $A_{\mathcal{N}}$ where $\mathcal{N} = \{1, ..., n\} \backslash \mathcal{B}$. By suitably permuting the columns of A we can re-express the standard form problem as

$$\begin{aligned} \min \quad & c_{\mathcal{B}}^T x + c_{\mathcal{N}}^T x \\ \text{s.t.} \quad & A_{\mathcal{B}} x_{\mathcal{B}} + A_{\mathcal{N}} x_{\mathcal{N}} = b \\ & x_{\mathcal{B}}, x_{\mathcal{N}} \geq 0 \end{aligned} \tag{7}$$

Let now $\hat{x} = \begin{pmatrix} \hat{x}_{\mathcal{B}} & \hat{x}_{\mathcal{N}} \end{pmatrix}^T = \begin{pmatrix} A_{\mathcal{B}}^{-1} b & 0 \end{pmatrix}^T$ be a basic feasible solution to the problem above. This solution is optimal if for all feasible x

$$\begin{aligned} c^T \hat{x} \leq c^T x &= c^T \hat{x} + (c_{\mathcal{N}} - c_{\mathcal{B}}^T A_{\mathcal{B}}^{-1} A_{\mathcal{N}})^T x_{\mathcal{N}} \\ 0 &\leq c_{\mathcal{N}} - c_{\mathcal{B}}^T A_{\mathcal{B}}^{-1} A_{\mathcal{N}} \end{aligned} \tag{8}$$

componentwise. Particularly, $\hat{x}$ is not optimal if

$$\exists\, j \in \mathcal{N}: \quad \mathrm{rc}(j) = c_j - c_{\mathcal{B}}^T A_{\mathcal{B}}^{-1} A^j < 0 \tag{9}$$

where $\mathrm{rc}(j)$ is commonly called *the reduced cost of variable* j. Because in practice, the matrix $A_{\mathcal{B}}^{-1}$ is rarely explicitly computable, the reduced cost is usually expressed in terms of an optimal dual solution which is a free byproduct of the simplex algorithm. The dual of problem (6) is given as

$$\begin{aligned} \max \quad & b^T \alpha \\ \text{s.t.} \quad & \alpha A \leq c \\ & \alpha \in \mathbb{R}^m. \end{aligned} \tag{10}$$

If $\hat{x}$ is an optimal basic feasible solution for (6), then $\hat{\alpha} = c_{\mathcal{B}} A_{\mathcal{B}}^{-1}$ is feasible because

$$\hat{\alpha} A = \hat{\alpha} \begin{pmatrix} A_{\mathcal{B}} & A_{\mathcal{N}} \end{pmatrix} = \begin{pmatrix} c_{\mathcal{B}} & c_{\mathcal{B}} A_{\mathcal{B}}^{-1} A_{\mathcal{N}} \end{pmatrix} \overset{(8)}{\leq} \begin{pmatrix} c_{\mathcal{B}} & c_{\mathcal{N}} \end{pmatrix} = c.$$

Also because the corresponding objective values are equivalent,

$$b^T\hat{\alpha} = b^T c_{\mathcal{B}} A_{\mathcal{B}}^{-1} = \begin{pmatrix} c_{\mathcal{B}}^T & c_{\mathcal{N}}^T \end{pmatrix} \begin{pmatrix} A_{\mathcal{B}}^{-1} b & 0 \end{pmatrix} = c^T \hat{x},$$

the strong duality implies that $\hat{\alpha}$ is an optimal solution for (10). Therefore (9) can be expressed as

$$\exists\, j \in \mathcal{N} : \quad \mathrm{rc}(j) = c_j - \hat{\alpha}^T A^j < 0.$$

The traditional, or primal, simplex algorithm uses reduced costs to guide the search for an optimal solution [37]. Starting from an initial basis, it identifies a nonbasic index j with a negative reduced cost and performs a pivot operation in which the corresponding A_j enters the basis, replacing some basic column A^i. This process is repeated until all nonbasic reduced costs are nonnegative. A major drawback of this method is the computation of reduced costs. When the problem is very large, explicitly scanning all nonbasic indices $\mathcal{N}$ may become computationally infeasible. This motivates alternative strategies, which are discussed next.

Column generation [11] is a space- and time-efficient alternative to the simplex algorithm that exploits the fact that many variables have a value of zero at the optimum and thus do need not to be included in the model. It starts with a smaller restriction of the original problem, with enough variables and columns to make the model feasible, and gradually increases its size by adding variables and associated columns that guarantee some improvement on its objective. That is, instead of working with the assumably huge set of $|J|$ variables, we start with a restriction of (6), called the *restricted master problem* (RMP)

$$\begin{aligned} \min \quad & \sum_{j \in J'} c_j x_j \\ \text{s.t.} \quad & \sum_{j \in J'} A^j x_j = b \qquad (\alpha) \\ & x_j \geq 0 \qquad\qquad \forall\, j \in J' \end{aligned}$$

where $J' \subseteq J$. At each iteration of the column generation algorithm, we solve the RMP, obtain a dual solution α, and solve a *pricing (sub)problem*

$$Z(PP) := \min\{c_j - \alpha^T A^j : j \in J\}. \tag{11}$$

If $Z(PP) < 0$ the variable x_j, its coefficient c_j and column A^j corresponding to a minimizer j are added to the RMP. The algorithm proceeds to the next iteration by solving the updated RMP until $Z(PP) \geq 0$, at which point it terminates and returns the solution of the current RMP, which is also optimal to master problem.

In almost every application the pricing problem (11) is not solved by an explicit enumeration over J, but rather implicitly over some domain X as

$$Z(PP) = \min\{f_0(x) - \alpha^T f_1(x) : x \in X\} \tag{12}$$

where $f = (f_0, f_1) : X \to \{(c_j, A^j)\}_{j \in J}$ defines a surjection between X and the pairs of objective coefficients and columns. In many applications, the domain X induces a structured combinatorial optimization problem, allowing (12) to be solved by a problem-specific combinatorial algorithm.

2.3 Dantzig-Wolfe decomposition for Integer Programs

Consider the following ILP, called as *the compact or original problem formulation*

$$\begin{aligned} Z(C) := \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & Dx \geq d \\ & x \in \mathbb{N}^n \end{aligned} \tag{13}$$

Consider now the subsystem $X = \{x \in \mathbb{N}^n : Dx \geq d\}$ of (13), and note that replacing X by $\text{conv}(X)$ does not change the integer optimum and yields an ILP-equivalent problem. Since $\text{conv}(X)$ is a polyhedron, the resolution theorem of Minkowski and Weyl [37] implies that every $x \in \text{conv}(X)$ can be written as

$$\forall\, x \in \text{conv}(X) : x = \sum_{p \in P} x_p \lambda_p + \sum_{r \in R} x_r \mu_r \tag{14}$$

where $\{x_p\}_{p \in P}$ and $\{x_r\}_{r \in R}$ are the finite collections of extreme points and rays, respectively, of $\text{conv}(X)$, $\lambda_p, \mu_r \geq 0$ and $\sum_{p \in P} \lambda_p = 1$. If X is bounded, the set of extreme rays is empty. Substituting (14) into (13) then yields an ILP-equivalent *extensive formulation*

$$Z(E_C) := \min \quad \sum_{p \in P} c^T x_p \lambda_p + \sum_{r \in R} c^T x_r \mu_r \tag{15.1}$$

$$\text{s.t.} \quad \sum_{p \in P} Ax_p \lambda_p + \sum_{r \in R} Ax_r \mu_r = b \qquad (\alpha) \tag{15.2}$$

$$\sum_{p \in P} \lambda_p = 1 \qquad (\beta) \tag{15.3}$$

$$\lambda_p \geq 0 \qquad \forall\, p \in P \tag{15.4}$$

$$\mu_r \geq 0 \qquad \forall\, r \in R \tag{15.5}$$

$$x = \sum_{p \in P} x_p \lambda_p + \sum_{r \in R} x_r \mu_r \tag{15.6}$$

$$x \in \mathbb{N}^n \tag{15.7}$$

which is often called the *Dantzig-Wolfe decomposition/reformulation* of (13).

In practice, the number of extreme points and rays is so large that solving (15) explicitly is intractable. Instead, we solve its linear relaxation using column generation and subsequently perform branching operations to obtain an integer solution. The linear relaxation is obtained by relaxing the integrality requirement on the variable x, which yields the linking constraint (15.6) unnecessary, leaving us with a master problem (15.1)-(15.5). After solving the restricted master, we obtain an optimal dual solution α, β and solve the pricing problem

$$Z(PP) := \min\{c_i - \alpha^T a_j - 1(i \in P)\beta : i \in P \cup R\} \tag{16}$$

where $1(\cdot)$ is the indicator function. Due to the obvious surjection $X \to \{(c^T x_j, Ax_j)\}_{j \in P \cup R}$, and β being constant, (16) can be given as an ILP

$$Z(PP) = \min\{c^T x - \alpha^T Ax - \beta : x \in X\}. \tag{17}$$

We can recognize when the minimizer of PP corresponds to an extreme point or ray by inspecting the objective value. If $Z(PP) < 0$ and finite, an optimal solution is an extreme point, and when $Z(PP) = -\infty$ solution corresponds to an extreme ray [11].

There are a couple of reasons why the extensive formulation is often preferred over the compact formulation. First, while both formulations are ILP-equivalent, they differ in their linear program LP-relaxations [45]. The extensive and compact LPs have feasible subdomains $\text{conv}(\{x \in \mathbb{N} : Dx \geq d\}) \subseteq \{x \in \mathbb{R}^n_{\geq 0} : Dx \geq d\}$, respectively, which means that during column generation with an extensive master, we solve a tighter relaxation and often obtain stronger lower bounds since $Z_{LP}(C) \leq Z_{LP}(E_C) \leq Z(C)$. Stronger bounds often mean that we need to solve fewer nodes in the branch-and-bound tree, as noted in the previous section.

Dantzig-Wolfe decomposition is also attractive from a branching perspective, because the pricing problem is defined over the same structured domain X as the compact variables. This property allows robust branching rules that do not disturb the column generation phase, and it is one of the main reasons why applying Dantzig-Wolfe decomposition is a standard practice in branch-and-price algorithms [45–47].

In addition, the matrix D in the compact formulation (13) and the subsystem X in the extensive formulation often have specific structures that can be utilized. For this thesis, the interesting case is when D allows a block diagonal format

$$D = \begin{pmatrix} D^1 & & & \\ & D^2 & & \\ & & \ddots & \\ & & & D^H \end{pmatrix}, \quad d = \begin{pmatrix} d^1 \\ d^2 \\ \vdots \\ d^H \end{pmatrix}$$

where $A = \begin{pmatrix} A^1 & A^2 & \cdots & A^H \end{pmatrix}$ and $c = \begin{pmatrix} c^1 & c^2 & \cdots & c^H \end{pmatrix}$ are divided accordingly. This allows the integer polyhedron X to decompose into H independent subsystems, $X = X^1 \times \cdots \times X^H$, where $X^h = \{x^h \in \mathbb{N}^{n_h} : D^h x^h \geq d^h\}$ for each $h \in \{1, ..., H\}$. Applying Dantzig-Wolfe decomposition to each subsystem, the relaxed extensive formulation, i.e., the master problem, takes the form

$$\begin{aligned}
\min \quad & \sum_{h=1}^{H} \sum_{p \in P^h} (c^h)^T x_p^h \lambda_p^h + \sum_{h=1}^{H} \sum_{r \in R^h} (c^h)^T x_r^h \mu_r^h & & \\
\text{s.t.} \quad & \sum_{h=1}^{H} \sum_{p \in P^h} A^h x_p^h \lambda_p^h + \sum_{h=1}^{H} \sum_{r \in R^h} A^h x_r^h \mu_r^h = b & & (\alpha) \\
& \sum_{p \in P^h} \lambda_p^h = 1 & \forall\, h \in \{1, ..., H\} & (\beta) \\
& \lambda_p^h \geq 0 & \forall\, h \in \{1, ..., H\},\ \forall p \in P^h & \\
& \mu_p^h \geq 0 & \forall\, h \in \{1, ..., H\},\ \forall p \in R^h &
\end{aligned} \tag{18}$$

where P^h and R^h index the extreme points and rays, respectively, of $\text{conv}(X^h)$. This then results to H independent pricing problems

$$Z(PP_h) = \min\{(c^h)^T x - \alpha^T A^h x - \beta_h : x \in X^h\}.$$

These subproblems are usually smaller and often have exploitable combinatorial structure, so each of them can be solved by specialized methods rather than by a generic MILP solver. Moreover, independent pricing problems can be processed in parallel, and one may even use a partial pricing, where only the most promising blocks are priced at a given iteration [11].

Additionally, if linking constraints and subsystems have identical characteristics, meaning $A^h = \tilde{A}, D^h = \tilde{D}, d^h = \tilde{d}$ and $c^h = \tilde{c}$ for all $h \in \{1, ..., H\}$, then $X^h = \tilde{X}, P^h = \tilde{P}$ and $R^h = \tilde{R}$ for every block h. By aggregating the variables, the blockwise master can be written as

$$\begin{aligned}
\min \quad & \sum_{p \in \tilde{P}} \tilde{c}^T x_p \lambda_p + \sum_{r \in \tilde{R}} \tilde{c}^T x_r \mu_r \\
\text{s.t.} \quad & \sum_{p \in \tilde{P}} \tilde{A} x_p \lambda_p + \sum_{r \in \tilde{R}} \tilde{A} x_r \mu_r = b \\
& \sum_{p \in \tilde{P}} \lambda_p = H \\
& \lambda_p \geq 0 && \forall\, p \in \tilde{P} \\
& \mu_r \geq 0 && \forall\, r \in \tilde{R}.
\end{aligned} \tag{19}$$

In the linear programming context, aggregation is a useful practice to simplify the model in applications with many interchangeable resources. It is also beneficial in integer linear programming, as it removes the symmetry caused by interchangeable subsystems, which has been shown to increase solution times [35]. However, in a branch-and-price context, which will be introduced in the next section, one must be careful when deciding whether to aggregate, as aggregated master variables may no longer correspond uniquely to the structured objects generated by pricing [12]. In addition, branching on aggregate variables is typically not sufficient to eliminate all fractional solutions and often cannot be enforced directly in the pricing problem [43, 46].

2.3.1 Dantzig-Wolfe decomposition of compact MFD formulation

Next, we apply the Dantzig-Wolfe decomposition technique to the compact arc-node formulation (4). This yields a problem both ILP- and LP-equivalent to the extensive arc-path formulation (5).

Let

$$X = \{(\{y_h\}_{h \in H}, \{x^h\}_{h \in H}, \{w_h\}_{h \in H}, \{z^h\}_{h \in H}) : (4.3) - (4.10)\} = X^1 \times \cdots \times X^H$$

be a set of feasible solutions of the compact arc-node formulation (4) and let $h \in \{1, ..., H\}$. If $y_h = 0$, then constraint (4.6) implies that $x^h = 0$ which results in $z^h = 0$ and $w_h = 0$ by (4.4) and (4.3), respectively. Alternatively, if $y_h = 1$ then using the same constraints implies that x^h is a $s - t$ path δ_p, $z^h = w_h \delta_p$ and

$w_h = \{1, ..., \min_{e \in p} f_e = f_p\}$. Therefore we can re-express X^h as

$$\begin{aligned} X^h &= \{(0,0,0,0)\} \cup \bigcup_{p \in P} \{(1, \delta_p, w, w\delta_p) : w = \{1, ..., f_p\}\} \\ &= \{(0,0,0,0)\} \cup \bigcup_{p \in P} X^h_p \end{aligned}$$

The convex hull of X^h_p is a line segment $\text{conv}(X^h_p) = \{(1, \delta_p, w, w\delta_p) : 1 \le w \le f_p\}$ having two extreme points $(1, \delta_p, 1, \delta_p)$ and $(1, \delta_p, f_p, f_p\delta_p)$. Correspondingly, the extreme points of $\text{conv}(X^h)$ are

$$\mathcal{P} = \{(0,0,0,0)\} \cup \{(1, \delta_p, 1, \delta_p) : p \in P\} \cup \{(1, \delta_p, f_p, f_p\delta_p) : p \in P\}.$$

By introducing variables $\lambda_{h0}, \{\lambda^0_{hp}\}_{p \in P}, \{\lambda^1_{hp}\}_{p \in P} \ge 0$ such that $\lambda_{h0} + \sum_{p \in P}(\lambda^0_{hp} + \lambda^1_{hp}) = 1$, we can express any $(y_h, x^h, w_h, z^h) \in \text{conv}(X^h)$ as

$$\begin{aligned} y_h &= \sum_{p \in P} (\lambda^0_{hp} + \lambda^1_{hp}), \qquad & x^h &= \sum_{p \in P} (\lambda^0_{hp} + \lambda^1_{hp})\delta_p, \\ w_h &= \sum_{p \in P} (\lambda^0_{hp} + f_p\lambda^1_{hp}), & z^h &= \sum_{p \in P} (\lambda^0_{hp} + f_p\lambda^1_{hp})\delta_p. \end{aligned} \tag{20}$$

By substituting these expressions in the place of the linking constraints and objective of the compact formulation, we obtain an ILP-equivalent extensive formulation

$$\begin{aligned} \min \quad & \sum_{h=1}^{H} \sum_{p \in P} (\lambda^0_{hp} + \lambda^1_{hp}) && && (21.1) \\ \text{s.t.} \quad & \sum_{h=1}^{H} \sum_{p \in P} (\lambda^0_{hp} + f_p\lambda^1_{hp})\delta_{ep} = f_e && \forall e \in E && (21.2) \\ & \sum_{p \in P} (\lambda^0_{hp} + \lambda^1_{hp}) \le 1 && \forall h \in \{1, ..., H\} && (21.3) \\ & \lambda^0_{hp} \ge 0 && \forall h \in \{1, ..., H\},\ \forall p \in P && (21.4) \\ & \lambda^1_{hp} \ge 0 && \forall h \in \{1, ..., H\},\ \forall p \in P && (21.5) \\ & \lambda^0_{hp} + \lambda^1_{hp} \in \{0, 1\} && \forall h \in \{1, ..., H\},\ \forall p \in P && (21.6) \\ & \lambda^0_{hp} + f_p\lambda^1_{hp} \in \mathbb{N} && \forall h \in \{1, ..., H\},\ \forall p \in P. && (21.7) \end{aligned}$$

Similarly as with the compact formulation, we have H slots, each of which is either unused ($\lambda^0_{hp} + \lambda^1_{hp} = 0$) or represents a single path used for decomposing the network flow ($\lambda^0_{hp} + \lambda^1_{hp} = 1$). The variables λ^0_{hp} and λ^1_{hp} together control the flow of path p by interpolating between the extreme values of 1 and f_p.

One can easily see the similarity between formulations (21) and (5) when we perform a change of variables $y_{hp} = \lambda^0_{hp} + \lambda^1_{hp}$ and $w_{hp} = \lambda^0_{hp} + f_p\lambda^1_{hp}$, and link them

via $w_p^h \leq f_p y_p^h$. This formulation then reads as

$$\min \quad \sum_{h=1}^{H} \sum_{p \in P} y_{hp} \tag{22.1}$$

$$\text{s.t.} \quad \sum_{h=1}^{H} \sum_{p \in P} \delta_{ep} w_{hp} = f_e \qquad \forall e \in E \tag{22.2}$$

$$w_{hp} \leq f_p y_{hp} \qquad \forall h \in \{1, ..., H\}, \forall p \in P \tag{22.3}$$

$$w_{hp} \geq y_{hp} \qquad \forall h \in \{1, ..., H\}, \forall p \in P \tag{22.4}$$

$$\sum_{p \in P} y_{hp} \leq 1 \qquad \forall h \in \{1, ..., H\} \tag{22.5}$$

$$y_{hp} \in \{0, 1\} \qquad \forall h \in \{1, ..., H\}, \forall p \in P \tag{22.6}$$

$$w_{hp} \in \mathbb{N} \qquad \forall h \in \{1, ..., H\}, \forall p \in P \tag{22.7}$$

and is both LP- and ILP equivalent to (21). If we further aggregate $y_p = \sum_{h=1}^{H} y_{hp}$ and $w_p = \sum_{h=1}^{H} w_{hp}$ this problem is also LP- and ILP equivalent to (5) where restriction $y_p \in \{0, 1\}$ is relaxed to $y_p \in \mathbb{N}$. Equivalence with $y_p \in \{0, 1\}$ version could be obtained by introducing constraints $\sum_{h \in H} y_{hp} \leq 1$ for each $p \in P$, but we omit those since the corresponding path-indexed duals would cause difficulties during the column generation phase. Also, any optimal solution will satisfy them nevertheless. We call both (21) and (22) as disaggregated extensive arc-path formulations, and while the latter has a more intuitive structure, the former is more suited for column generation.

To conclude this section, we show that the formulation (22) is indeed at least as strong as the compact formulation (4), and for some MFD instances, it is even strictly stronger. Consequently, formulations (21) and (5) are stronger as well.

Proposition 1.
For every MFD instance, the LP relaxed domain of formulation (22), projected onto the compact-variable space, is contained in the LP relaxed domain of formulation (4). Moreover, this inclusion is strict for some instances.
Proof:
Consider the polyhedron representing the projection from the extensive formulation domain to the compact formulation domain

$$P = \{(\tilde{y}, \tilde{x}, \tilde{w}, \tilde{z}) : (22.2) - (22.5), y_{hp} \in [0, 1], w_{hp} \geq 0, \\ \tilde{y}_h = \sum_{p \in P} y_{hp}, \tilde{x}_e^h = \sum_{p \in P} \delta_{ep} y_{hp}, \tilde{w}_h = \sum_{p \in P} w_{hp}, \tilde{z}_e^h = \sum_{p \in P} \delta_{ep} w_{hp}\}.$$

If $(\tilde{y}, \tilde{x}, \tilde{w}, \tilde{z}) \in P$, then it satisfies the compact formulation constraints (4.2)-(4.6). First, constraint (22.2) directly implies (4.2). Since every δ_p is the incidence vector of a $s-t$ path, the projected variables $\tilde{x}^h$ and $\tilde{z}^h$ satisfy the flow-conservation constraints (4.6) and (4.3). Moreover, by (22.3) and the fact that $f_p \leq f_e$ whenever $e \in p$

$$\tilde{z}_e^h = \sum_{p \in P} \delta_{ep} w_{hp} \leq \sum_{p \in P} \delta_{ep} f_p y_{hp} \leq f_e \sum_{p \in P} \delta_{ep} y_{hp} = f_e \tilde{x}_e^h. \tag{23}$$

Thus (4.4) holds. Similarly, (22.5) implies

$$\tilde{w}_h = \sum_{p \in P} w_{hp} \geq \sum_{p \in P} y_{hp} = \tilde{y}_h$$

so (4.5) holds. Finally, (22.5) implies $0 \leq \tilde{y}_h \leq 1$ *and* $0 \leq \tilde{x}_e^h \leq 1$*. Therefore, every point in the projected LP relaxation of (22) is feasible for the LP relaxation of (4).*

However, for some flow network instances, there may exist LP-feasible compact formulation solutions that cannot be reconstructed using the solutions in the LP-feasible extensive formulation domain. This happens because the last inequality in (23) may be strict in some occasions. As an example case, consider the Figure 2 which illustrates a feasible solution of the LP relaxation of the compact formulation for the given network instance. Suppose now that there exists a point $(\tilde{y}, \tilde{x}, \tilde{w}, \tilde{z}) \in P$ *that is equivalent to this solution. Because* $\tilde{z}_{e_3}^1 = w_{1p_3} + w_{1p_4} = 0$ *this would require that*

$$w_{1p_1} = \tilde{z}_{e_4}^1 = \frac{40}{9} \quad \textit{and} \quad w_{1p_2} = \tilde{z}_{e_6}^1 = \frac{5}{9}.$$

Because constraint (22.3) gives $\frac{w_{1p}}{f_p} \leq y_{1p}$*, while (22.5) gives* $\sum_{p \in P} y_{1p} \leq 1$*, together they would imply*

$$1 \geq \sum_{p \in P} y_{1p} \geq \sum_{p \in P} \frac{w_{1p}}{f_p} \geq \frac{w_{1p_1}}{f_{p_1}} + \frac{w_{1p_2}}{f_{p_2}} = \frac{40/9}{5} + \frac{5/9}{1} = \frac{13}{9}$$

which is impossible. Therefore, no feasible solution of the extensive LP relaxation can be projected onto the compact solution in Figure 2. □

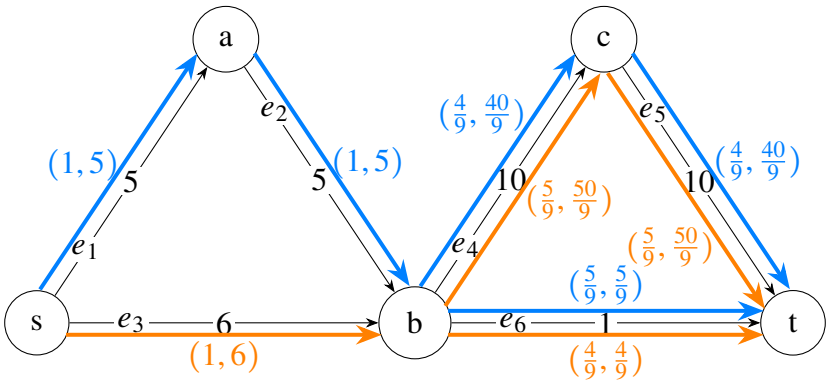


Figure 2: An example of a solution to the LP-relaxed compact formulation that is not the projection of any solution of the extensive formulation. The black arrows represent the original flow network having paths $p_1 = (e_1, e_2, e_4, e_5)$, $p_2 = (e_1, e_2, e_6)$, $p_3 = (e_3, e_4, e_5)$, $p_4 = (e_3, e_6)$ with bottlenecks $f_{p_1} = 5$, $f_{p_2} = 1$, $f_{p_3} = 6$, $f_{p_4} = 1$, respectively, and decomposition upper bound $H \geq 3$. The Blue arrows illustrate the fractional solution pair (x^1, z^1), and orange arrows illustrate the fractional solution pair (x^2, z^2), of the compact formulation.

2.4 Branch-and-price

Branch-and-price is a solution framework for large-scale integer programming problems that combines branch-and-bound with column generation [10]. In this framework, each node of the branch-and-bound tree is solved via column generation, and branching decisions are later imposed on the master and/or pricing problems to enforce integrality in the long run. While the concept is simple, one must pay close attention to the structures on which branching is performed since the branching decisions often change the master problem formulations of the child nodes. As a result, the pricing problem in the column generation phase may become increasingly more complex for nodes deeper in the branch-and-bound tree. In this section, we introduce the branch-and-price solution method for large ILPs in the general setting. Specifically, we focus on two high-level branching strategies: branching on the master problem variables and branching on the original problem variables, that is, the variables of the pricing problem.

Before describing the two general branching strategies, we briefly explain what branching means and what it tries to accomplish in a conventional ILP setting. First, the LP-relaxation is solved, and if the solution does not violate the integrality restrictions, there is nothing to do. Otherwise, one must divide, or branch, the current LP-relaxation domain into two or more subdomains so that the current fractional solution is excluded from each of them. In addition, every integer-feasible solution must be included in the union of the resulting subdomains. If the division operation satisfies these two requirements, it is called a *valid branching rule*. Starting from the LP-relaxation of the initial unmodified ILP, the repeated application of a valid branching rule produces a hierarchy of smaller LP problems that can be identified as nodes in a search tree. The initial LP-relaxation then corresponds to the root node of the tree. Finally, if the repeated use of a valid branching rule can eliminate every fractional solution while keeping the search tree finite, we call it a *complete branching rule*. [10]

2.4.1 Branching on the master variables

Consider the root node master problem obtained by relaxing the integrality requirements in the extensive formulation (15) and assume for simplicity that $R = \varnothing$. Solving this problem by column generation yields an optimal solution $\hat{\lambda}$. Although the integrality requirements in (15) are imposed on the reconstructed variable x, it is natural to ask whether branching can be carried out directly in the master-variable space, as in the extensive MFD formulation (22) or in the set partitioning problem (24) below.

A natural first idea would be to branch directly on a fractional component $\hat{\lambda}_p \in (0, 1)$ and create two child nodes by fixing either $\lambda_p = 0$ or $\lambda_p = 1$. However, this simple branching strategy often proves to be inefficient while the algorithm proceeds. In a binary case, the branch $\lambda_p = 0$ is usually much weaker compared to $\lambda_p = 1$ which may lead to an unbalanced branch-and-bound tree. Forbidding the use of a single column p out of an exponentially many candidates does not often yield any significant restrictions on the problem domain, resulting in only small bound improvements. This can then lead to slow convergence and deep branches on the search tree. In contrast,

forcing the use of a column is usually a much stronger restriction because it forces the inclusion of a specific combinatorial pattern in a solution. As a result, this branch often produces significant bound improvements or quickly proves itself infeasible. [10, 46]

Another issue with single master variable branching is that it does not usually correspond to a natural structural restriction on the pricing domain X. For example, in integer multicommodity flow problems, where the pricing problem is given as a shortest path problem [4], branching directly on the path-indicator master variables would force a commodity either to use a particular path or to exclude that path entirely. The latter decision is difficult to impose within the shortest path problem, as solving the pricing problem may again return the forbidden path. Therefore, enforcing this branching decision may require a next-shortest-path procedure, whose running time can grow linearly with the depth of the branch-and-bound tree. Similarly, when the pricing problem is implemented as a general MILP problem, such as the knapsack problem in [9], additional constraints may be needed to prevent the re-generation of previously forbidden columns.

The drawbacks mentioned above motivate the search for alternative methods for master variable branching. The oldest known general rule was proposed by Ryan and Foster [36] that works on the special case of set partitioning problems. The general set partitioning problem is given as

$$\begin{aligned} \min \quad & \sum_{p \in P} c_p \lambda_p \\ \text{s.t.} \quad & \sum_{p \in P} a_{ip} \lambda_p = 1 \qquad \forall\, i \in I \\ & \lambda_p \in \{0,1\} \qquad \forall\, p \in P \end{aligned} \tag{24}$$

where binary coefficient $a_{ip} = 1$ if item i belongs to the set indexed by p. For example, in vehicle routing problems [10], $a_{ip} = 1$ represents that a customer i is visited on route p, whereas in crew scheduling problems [44], it may indicate that flight i is included in pairing p. Instead of branching directly on fractional λ_p values, Ryan and Foster proposed branching on pairs of items $i, j \in I$. For a current LP solution $\hat{\lambda}$, define

$$\theta_{ij} = \sum_{\substack{p \in P: \\ a_{ip} = a_{jp} = 1}} \hat{\lambda}_p$$

which is the total LP weight assigned to sets containing both items i and j. Because each item must be covered by a set exactly once, every integer-feasible solution satisfies $\theta \in \{0,1\}$. In particular, $\theta_{ij} = 1$ if both items belong to the same selected set, and $\theta_{ij} = 0$ if they are covered by different sets. Therefore, whenever $0 < \theta_{ij} < 1$ the current solution is fractional and can be eliminated by branching on these two alternatives. The first branch forbids the sets containing only one of items i and j, which leads to a restricted pricing domain $\{p \in P : a_{ip} = a_{jp}\}$. In the second branch, the sets containing both i and j are forbidden, which gives a domain $\{p \in P : a_{ip} + a_{jp} \leq 1\}$. These restrictions are often relatively easy to enforce in the pricing problem [5].

The Ryan–Foster branching rule has later been generalized to broader classes of problems whose master problems do not have a strict set partitioning structure. Vanderbeck [45] proposed a general strategy in which branching is performed on the fractional sum of the master variables. In particular, they observed that for any fractional master solution $\{\hat{\lambda}_p\}_{p\in P}$ there exists a subset $\hat{P} \subseteq P$ whose total weight $\hat{w} = \sum_{p\in\hat{P}} \hat{\lambda}_p$is fractional and the branching takes the form

$$\sum_{p\in\hat{P}} \lambda_p \leq \lfloor \hat{w} \rfloor \quad \text{or} \quad \sum_{p\in\hat{P}} \lambda_p \geq \lceil \hat{w} \rceil. \tag{25}$$

Branching on an individual master variable λ_p is a special case where $\hat{P} = \{p\}$. Vanderbeck then proposed two different ways to define $\hat{P}$. In the first approach, given a fractional solution $\hat{\lambda}$ of (15) where $R = \varnothing$, there exists a hyperplane $(y, y_0) \in \mathbb{Z}^{n+1}$ such that

$$\sum_{\substack{p\in P:\\ y^T x_p \geq y_0}} \hat{\lambda}_p$$

is fractional. Here x_p is an extreme point of $\operatorname{conv}(X)$.

In the second approach, $\hat{P}$ is defined in terms of lower and upper bounds on some components of x_p. When denoting a component lower bound constraint $x_{pi} \geq v$ as $\mathfrak{b} \equiv (i, \geq, v)$ Vanderbeck defined the column class satisfying this constraint as $P(\mathfrak{b}) = \{x_p \in X : x_{pi} \geq v\}$. Similarly, the component upper bound constraint $x_{pi} < v$ is expressed as $\mathfrak{b}^c \equiv (i, <, v)$ which results to column class $P(\mathfrak{b}^c) = \{x_p \in X : x_{pi} < v\}$. These column classes yield a partition $P = P(\mathfrak{b}) \cup P(\mathfrak{b}^c)$, and for a set of component lower and upper bounds $\mathfrak{B}$ we have $P(\mathfrak{B}) = \bigcap_{\mathfrak{b}\in\mathfrak{B}} P(\mathfrak{b})$. To use component bounds as branching constraints to cut off a fractional master solution, Vanderbeck shows that given a fractional solution $\hat{\lambda}$ with fractionality $f = \sum_{p\in P}(\hat{\lambda}_p - \lfloor \hat{\lambda}_p \rfloor)$, there exists a set of component bounds $\mathfrak{B}$ with $|\mathfrak{B}| \leq \lfloor \log f \rfloor + 1$ such that

$$\sum_{p\in P(\mathfrak{B})} \hat{\lambda}_p$$

is fractional.

2.4.2 Branching on the original variables

An alternative strategy is to branch on the components of the original x variables if the decomposition map (14) is available. This is often relatively straightforward to include into the branch-and-price pipeline, as the resulting branching decisions can be modeled as an explicit constraint rows in the master problem. Therefore, only the objective of the pricing problem needs to be modified, while the domain often stays unaltered [12].

Suppose now we have a solution for (15) such that reconstructed $\hat{x}$ has a fractional component $\hat{x}_i$, and to eliminate it, we branch by adding constraint

$$\sum_{p\in P} x_{pi}\lambda_p + \sum_{r\in R} x_{ri}\mu_r \leq \lfloor \hat{x}_i \rfloor, \quad \text{or} \quad \sum_{p\in P} x_{pi}\lambda_p + \sum_{r\in R} x_{ri}\mu_r \geq \lceil \hat{x}_i \rceil \tag{26}$$

directly to the master problem. These new constraints would then have corresponding dual variables γ_i^0 and γ_i^1, respectively, and only the objective of the pricing problem needs to be modified. For example, shortest-path pricing remains a shortest-path problem where the extra dual only changes the weight of a single arc in the graph [4].

Alternatively, the branching rule (26) can be imposed in the pricing problem. For the "≤"-branch ("≥"-branch handled analogously), one must first manually eliminate the master variables violating the rule, which can be done by removing, or fixing to zero, all variables λ_p and μ_r for which $x_{pi} > \lfloor \hat{x}_i \rfloor$ and $x_{ri} > \lfloor \hat{x}_i \rfloor$, respectively. The objective of the pricing problem (17) stays the same, but the domain is changed from X to $\{x \in X : x_i \leq \lfloor \hat{x}_i \rfloor\}$. If this restricted pricing problem remains tractable, this option is preferable because it yields an LP relaxation at least as strong as the master-enforced alternative, since

$$\text{conv}(\{x \in X : x_i \leq \lfloor \hat{x}_i \rfloor\}) \subseteq \{x \in \text{conv}(X) : x_i \leq \lfloor \hat{x}_i \rfloor\}.$$

Barnhart et al. [4] proposed this kind of rule for origin-destination integer multicommodity flow problems. The standard master variable branching is poorly suited for their setting because fixing a path variable or requiring the commodity to use a specific arc can destroy the shortest path structure of the pricing problem. Instead, when the commodity is fractionally assigned to multiple paths, they take two of such paths and identify the first divergence node d, from which they split via arcs a_1 and a_2. Because every integer-feasible solution must use at most one outgoing arc from d, and the current fractional solution uses both a_1 and a_2, it is possible to cut off the current fractional solution by partitioning the $\text{out}(d)$ into two subsets, one containing a_1 and the other containing a_2. Two child nodes are then created, where the commodity is forbidden to use the arcs in one of the resulting subsets. The details of this branching strategy are postponed to Section 3.2, where we illustrate its applicability to the MFD problem.

Lastly, we consider the case of branching on the original variables when the master problem variables are aggregated over independent and identical subsystems, as in (19). In this context, the aggregation prevents us from identifying the original slot indices available in the disaggregated model (18), which prevents the straight access to the original variables. Specifically, for the disaggregated model, the original variables x^h are available via an unique projection

$$x^h = \sum_{p \in P^h} x_p^h \lambda_p^h + \sum_{r \in R^h} x_r^h \mu_r^h$$

but after aggregation, we have only access to aggregated original variables via projection

$$\sum_{h=1}^{H} x^h = \sum_{p \in \tilde{P}} x_p \lambda_p + \sum_{r \in \tilde{R}} x_r \mu_r,$$

and there is no unique mapping back to disaggregated original variables. Therefore, at best, one may branch on aggregated original quantities. The downside is that such a branching rules are often too weak, because the integrality of the aggregated original

solution does not necessarily imply the integrality of the disaggregated solution. In addition, branching by modifying the bounds of aggregated original variables often gives only a little improvement on the dual bound [12, 48]. Therefore, if branching on original variables is needed while solving an aggregated master problem, it is often necessary to reintroduce disaggregated or auxiliary variables using artificial methods [12, 46].

3 Solving MFD via branch-and-price

In this section, we propose a complete branch-and-price pipeline for the MFD problem. First, we describe how to obtain an initial feasible solution. We then derive the reduced cost formulas and formulate the pricing problem used during column generation. Finally, we propose and compare suitable branching strategies.

The first step of the branch-and-price algorithm is to obtain an initial feasible solution, which is required to ensure that the restricted master problem is feasible in the first iteration of column generation. This provides a valid starting basis and dual information needed to price in new columns. An initial feasible solution for the MFD problem can be obtained using the following algorithm, which uses the greedy shortest-path heuristic of [49].

Algorithm 1 Feasible minimum flow decomposition [49]

Require: Flow network $G = (V, E, f)$.

1: Initialize residual flow vector $r = f$
2: **while** $G = (V, \{e \in E : r_e > 0\})$ has at least one $s - t$ path do **do**
3: Find a shortest $s - t$ path p in G.
4: $r_p = \min_{e \in p} r_e$
5: Save path p and its flow r_p.
6: Update $r_e \leftarrow r_e - 1(e \in p) r_p$.

At each iteration of Algorithm 1, a shortest path p is extracted from the unit-weight graph G whose active arcs are those having positive residual flow. The flow assigned to p is the bottleneck residual flow over its arcs. The resulting path-flow pair (p, r_p) then corresponds to one element in a decomposition (P, w) (see Definition 2). Lastly, we compute the remaining network flow needed to be decomposed by subtracting the path flow r_p from the arc weights, which are affected by p. For each affected node v, the updated in and out flows, $\text{flow}_{\text{in}}(v) - r_p$ and $\text{flow}_{\text{out}}(v) - r_p$, still satisfy the flow conservation property (1). Furthermore, at each iteration, updating the arc weights causes at least one arc to have a zero flow, causing the algorithm to terminate after a finite number of iterations. The extraction of a shortest path in line 3 is only a heuristic and can be replaced by any $s - t$ path selection rule. The effects of different path selection rules on solution quality are discussed in [49].

3.1 Column generation

Previously, we derived three equivalent extensive formulations for the MFD problem. We now select the LP-relaxation of one of these formulations as the master problem. The first formulation introduced, formulation (5), can be interpreted as an aggregation of the Dantzig-Wolfe reformulation (22). However, this aggregation removes part of the disaggregated structure that is useful for branching. In particular, the variables y_p and w_p describe only the total use of a path p, but not identify which slot position

realizes that use. Hence, although formulation (5) is a valid extensive formulation, it is not the most suitable master problem for branch-and-price.

We also discard the disaggregated formulation (22) with explicit path-indicator and flow variables because of its column-dependent constraints (22.3). During the practical column generation phase, reduced-cost computation requires dual variables associated with all constraints. However, if the restricted master problem contains only an active column set $P' \subset P$, then the dual values are available only for the rows currently present in the restricted problem. As a result, the reduced cost of a candidate entering column cannot be computed exactly when it depends on the dual values associated with column dependent rows that are not yet present. The issue of column dependent rows has received more attention recently [26, 40], and the proposed solution procedures typically rely on generating both columns and rows, either alternately or simultaneously. These methods are out of scope of this thesis. Therefore, we proceed with the extensive disaggregated formulation (21), which supports robust branching rules and fits naturally within the column generation framework.

As a final remark before deriving the pricing problem, we note that the disaggregated extended formulation (21) is highly symmetric, meaning that many integer solutions are equivalent up to a permutation. Such symmetry may cause the branch-and-bound search to revisit structurally identical nodes, leading to redundant exploration and, consequently, weakening the computational performance of branch-and-price [35]. To partially reduce this symmetry, we impose additional symmetry-breaking constraints that order the path variables according to the amount of flow they carry. Specifically, we require that the path in position h carries at least as much flow as the path in position $h+1$. This does not eliminate all symmetry since if several paths in an optimal solution carry the same amount of flow, they may be permuted while still yielding equivalent solutions. After adding these variable-ordering constraints to the disaggregated extensive formulation, the symmetry-breaking master problem formulation at the root node reads as

$$\min \quad \sum_{h=1}^{H} \sum_{p \in P} (\lambda^0_{hp} + \lambda^1_{hp}) \tag{27.1}$$

$$\text{s.t.} \quad \sum_{h=1}^{H} \sum_{p \in P} (\lambda^0_{hp} + f_p \lambda^1_{hp}) \delta_{ep} = f_e \qquad \forall e \in E \quad (\alpha) \tag{27.2}$$

$$\sum_{p \in P} (\lambda^0_{hp} + \lambda^1_{hp}) \leq 1 \qquad \forall h \in \{1, ..., H\} \quad (\beta) \tag{27.3}$$

$$\sum_{p \in P} (\lambda^0_{(h+1)p} + f_p \lambda^1_{(h+1)p}) - (\lambda^0_{hp} + f_p \lambda^1_{hp}) \leq 0 \qquad \forall h \in \{1, ..., H-1\} \quad (\gamma) \tag{27.4}$$

$$\lambda^0_{hp} \geq 0 \qquad \forall h \in \{1, ..., H\}, \forall p \in P \tag{27.5}$$

$$\lambda^1_{hp} \geq 0 \qquad \forall h \in \{1, ..., H\}, \forall p \in P \tag{27.6}$$

where α, β and γ denote the dual variable vectors corresponding to their respective constraints.

Let $h \in \{1, ..., H\}$. Since the master problem consist of two types of variables λ^0_{hp}

and λ^1_{hp}, their corresponding reduced costs are

$$\mathrm{rc}_0(h,p) = 1 - \sum_{e\in E} \delta_{ep}\alpha_e - \beta_h - 1(h \geq 2)\gamma_{h-1} + 1(h \leq H-1)\gamma_h \tag{28}$$

and

$$\mathrm{rc}_1(h,p) = 1 - f_p \sum_{e\in E} \delta_{ep}\alpha_e - \beta_h - f_p 1(h \geq 2)\gamma_{h-1} + f_p 1(h \leq H-1)\gamma_h \tag{29}$$

respectively.

For the columns associated with variables λ^0_{hp}, the corresponding pricing subproblem PP^0_h reads as

$$Z(PP^0_h) = \min_{p\in P} \sum_{e\in E} \delta_{ep}(-\alpha_e) - b_h \tag{30}$$

where $b_h = 1(h \geq 2)\gamma_{h-1} - 1(h \leq H-1)\gamma_h$. If $Z(PP^0_h) < \beta_h - 1$, the corresponding minimizer $p \in P$ decides the entering column δ_p for slot h. Specifically, given a fixed h, the summation term is the length of the path p over the original graph G, but where the arc weights are given by $-\alpha$. Therefore, PP^0_h is a standard shortest path problem with a shift b_h applied to all path lengths in slot h.

For the columns associated with variables λ^1_{hp}, the corresponding pricing subproblem PP^1_h reads as

$$Z(PP^1_h) = \min_{p\in P} f_p\Big(\sum_{e\in E} \delta_{ep}(-\alpha_e) - b_h\Big). \tag{31}$$

If $Z(PP^1_h) < \beta_h - 1$, the corresponding minimizer $p \in P$ determines the entering column δ_p for slot h. If f_p is replaced in the extensive formulation by a common upper bound, for example $f_p = \max_{e\in E} f_e$, for every $p \in P$, the resulting ILP remains equivalent to (21) and PP^1_h reduces to a standard shortest path problem. However, using a universal bound like this can significantly weaken the LP relaxation of the extensive formulation. For instance, for the network shown in Figure 2, the resulting extensive LP relaxation yields a dual bound of $\frac{11}{6}$, which is smaller than the bound of 2.0 obtained from the compact formulation.

If we wish to keep $f_p = \min_{e\in p} f_e$ in the formulation and preserve the strength of the formulation, an alternative approach is required. In this form, the pricing problem is no longer a standard shortest path problem because its objective is not additive over arcs. However, we can re-express PP^1_h as

$$Z(PP^1_h) = \min_{c\in f_{\mathrm{cap}}} c \min_{\substack{p\in P \\ \text{s.t. } f_p = c}} \sum_{e\in E} \delta_{ep}(-\alpha_e) - b_h \tag{32}$$

where $f_{\mathrm{cap}} = \{f_e : e \in E\}$ is the set of distinct arc-flow values. In particular, every possible path bottleneck f_p belongs to f_{cap}. Therefore, we can iterate over all candidate bottleneck values $c \in f_{\mathrm{cap}}$, solve a shortest path problem over the $s-t$ paths whose bottleneck flow is exactly c, and take the overall minimum. Because G is assumed to

be a DAG, this procedure can be implemented as a dynamic programming algorithm given below.

Algorithm 2 Pricing problem computation for slot $h \in \{1, ..., H\}$

Require: Optimal dual solution (α, β, γ), flow network $G = (V, E, f)$,

1: Initialize $p^* = \varnothing$, $f_{p^*} = 0$, $\text{rc}^* = \infty$ and let $f_{\text{cap}} = \{f_e : e \in E\}$ be the set of arc capacities, without duplicates, representing all possible path bottleneck values.
2: **for all** $c \in f_{\text{cap}}$ **do**
3: Let $E' = \{e \in E : f_e \geq c\}$, $\alpha' = \{\alpha_e : e \in E'\}$ and $G' = (V, E', \alpha')$. Sort G' in topological order
4: Initialize forward DP table: $D_F[s] = 0$ and $D_F[v] = \infty$ for $s \neq v \in V$
5: **for all** $v \in V$ in topological order **do**
6: $D_F[v] = \min\{D_F[u] + (-\alpha'_e) : e = (u, v) \in E'\}$
7: Initialize backward DP table: $D_B[t] = 0$ and $D_B[v] = \infty$ for $t \neq v \in V$
8: **for all** $v \in V$ in reverse topological order **do**
9: $D_B[v] = \min\{D_B[u] + (-\alpha'_e) : e = (v, u) \in E'\}$
10: Find the length of a shortest path from s to t having capacity c by computing $l_{st} = \min\{D_F[u] + (-\alpha'_e) + D_B[v] : e = (u, v) \in E' \text{ s.t. } f_e = c\}$. Reconstruct the corresponding path p
11: Given p and l_{st}, compute the reduced cost $\text{rc}_1(h, p)$
12: **if** $\text{rc}_1(h, p) < \text{rc}^*$ **then**
13: $p^* \leftarrow p$
14: $f_{p^*} \leftarrow c$
15: $\text{rc}^* \leftarrow \text{rc}_1(h, p)$
16: **return** $(p^*, f_{p^*}, \text{rc}^*)$

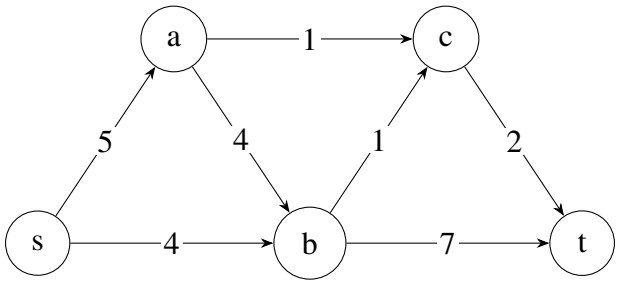
For each fixed bottleneck value c, Algorithm 2 first constructs a graph G' containing only arcs having weight values at least c. Therefore, any path with an exact bottleneck value of c is still contained in G'. On the other hand, no path with a bottleneck value less than c is contained in G'. Therefore, to find a shortest $s - t$ path from G with a bottleneck value of exactly c, it is sufficient to find a shortest $s - t$ path from G' that uses at least one arc with a weight of c.

Furthermore, because G' is a subgraph of the acyclic graph G, it is also acyclic and admits a topological ordering of nodes. Because acyclicity also disallows the presence of negative cycles in a graph, the shortest distances from node s to any other node can then be computed by a forward dynamic programming pass over this node ordering [1]. The shortest distances from each node to node t can be similarly computed by performing the dynamic programming pass in reverse order. These computations are performed on lines 4-9.

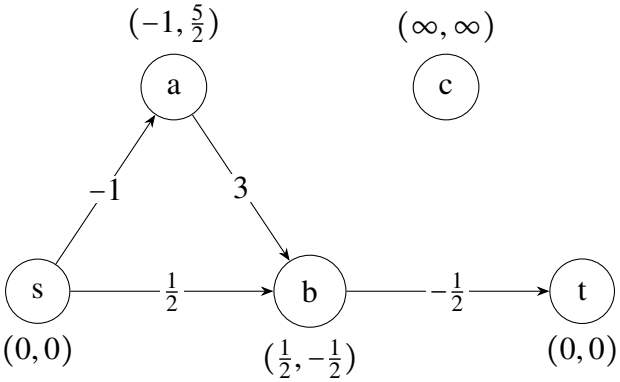
On line 10, $D_F[u]$ and $D_B[v]$ are the lengths of shortest paths from s to u and v to t, respectively. For a fixed $e = (u, v) \in E'$, the value of $D_F[u] - \alpha'_e + D_B[v]$ is then the length of a shortest path restricted to using the arc e. Minimizing this quantity over all arcs having a weight of c then gives the length of a shortest $s - t$ path having an exact

bottleneck value of c. After reconstructing the corresponding path, we compute its reduced cost using the formula (29) with $f_p = c$. Repeating this procedure for each possible bottleneck value and keeping track of a path having a smallest reduced cost then yields an optimal solution for the pricing problem PP_h^1. The algorithm operation is illustrated in Figure 3.

Given a fixed bottleneck value c, the graph G' can be constructed in $O(|E|)$ time. The forward and backward shortest path computations take $O(|V|+|E|)$ time each. The scan over arcs satisfying $f_e = c$ take $O(|E|)$ time. By performing these computations for each distinct bottleneck value, the total running time of the algorithm results in $O(|f_{\text{cap}}|(|V| + |E|))$.



(a) Original flow network with $f_{\text{cap}} = \{1, 2, 4, 5, 7\}$



(b) Network G' on iteration $c = 4$ where the arc weights are given by a made-up α. The dynamic programming tables are given as pairs $(D_F[u], D_B[u])$ next to a node. The length of a shortest path with $f_p = 4$ is computed as $\min\{-1 + 3 - \frac{1}{2}, 0 + \frac{1}{2} - \frac{1}{2}\}$.

Figure 3: Illustration of Algorithm 2

3.2 Branching

In this section, we propose branching strategies within the branch-and-price framework for the MFD problem. We remind that the solution of the master problem (27.1)-(27.6) is considered integer-valued if either $\lambda_{hp}^0 + \lambda_{hp}^1 \in \{0, 1\}$ and $\lambda_{hp}^0 + f_p \lambda_{hp}^1 \in \mathbb{N}$ for every

$h \in \{1, ..., H\}$ and $p \in P$, or alternatively, if the reconstructed compact variables via maps (20) are integer-valued. Otherwise, the solution is considered fractional.

As mentioned in Section 2.4, the branch-and-bound strategy of branching directly on master variables, or in our case, on quantities $\lambda^0_{hp} + \lambda^1_{hp} \in \{0, 1\}$ and $\lambda^0_{hp} + f_p \lambda^1_{hp} \in \mathbb{N}$, is often difficult to handle at the pricing problem level. Such branching decisions may lead to unbalanced branch-and-bound trees, which may negatively affect the performance of the overall algorithm. We first illustrate the difficulties that may arise if this branching strategy is applied to the MFD master problem.

Suppose that, for some $h \in \{1, ..., H\}$ and $p \in P$, the value of $\lambda^0_{hp} + \lambda^1_{hp}$ is fractional. A direct branching rule would create two subproblems by imposing $\lambda^0_{hp} + \lambda^1_{hp} = 0$ or $\lambda^0_{hp} + \lambda^1_{hp} = 1$, respectively. The latter branch forces slot h to use path p, thereby significantly restricting the feasible region and potentially producing a strong improvement in the dual bound. In contrast, the former branch only forbids one path among exponentially many possible paths. As a result, it barely restricts the feasible region and is unlikely to improve the dual bound significantly. Repeated use of such branching decisions may therefore produce a highly unbalanced search tree.

This strategy would also complicate column generation at the deeper nodes of the tree. At depth k, one would have to maintain a set for up to k forbidden paths for each slot h and ensure that the pricing problem does not regenerate these paths. Consequently, Algorithm 2 would have to be modified so that it searches for a feasible path that is not among the forbidden ones. This increases both the structural and time-wise complexity of the pricing problem. Therefore, rather than forcing or forbidding complete paths, it is preferable to guide the search toward integral solutions by imposing structural branching decisions at the arc level.

The branching method must remove the fractionality at three levels: slot activation, slot flow, and path choice. We implement this on the original variables induced by a master solution rather than directly on the individual master variables.

We first consider the fractionality in the slot activation. Recall that for a solution to be integer-feasible for the disaggregated extensive formulation (21.1)-(21.7), each slot $h \in \{1, ..., H\}$ must be either active or inactive. An active slot represents a single path, whereas an inactive slot does not contribute to the decomposition. However, in an LP-relaxed master problem solution, a slot may be only fractionally active. That is, there may exist a slot h such that $0 < y_h = \sum_{p \in P} (\lambda^0_{hp} + \lambda^1_{hp}) < 1$. This type of fractionality can be eliminated by branching on the activation status of slot h, using the following two constraints

$$(L) : y_h = \sum_{p \in P} (\lambda^0_{hp} + \lambda^1_{hp}) = 0 \quad \text{or} \quad (R) : y_h = \sum_{p \in P} (\lambda^0_{hp} + \lambda^1_{hp}) = 1. \tag{33}$$

In practice, constraint (L) can be implemented by fixing $\lambda^0_{hp} = \lambda^1_{hp} = 0$ for all $p \in P$. This rule, together with flow-ordering constraints then gives

$$\begin{aligned} y_h = 0 &\implies w_h = \sum_{p \in P} (\lambda^0_{hp} + f_p \lambda^1_{hp}) = 0 \\ &\implies w_k = 0, \ \forall k \geq h \\ &\implies y_k = 0, \ \forall k \geq h, \end{aligned}$$

i.e. $\lambda^0_{kp} = \lambda^1_{kp} = 0$ for all $k \geq h$ and $p \in P$. In this case, we call the slots $h, ..., H$ inactive, which tells us the size K of any integer-feasible decomposition on the resulting subtree must satisfy $K \leq h$.

Constraint (R), on the other hand, can be implemented by replacing the corresponding inequality constraint in (27.3) with an equality. This rule, together with flow-ordering constraints then give

$$\begin{aligned} y_h = 1 &\implies w_h = \sum_{p \in P} (\lambda^0_{hp} + f_p \lambda^1_{hp}) \geq 1 \\ &\implies w_k \geq 1, \ \forall k \leq h \\ &\implies y_k > 0, \ \forall k \leq h. \end{aligned}$$

Therefore, all slots $k \leq h$ must be active in any integer-feasible solution, and it is safe to strengthen the (R) branch by fixing $y_k = 1$ for all $k \leq h$ even though they are not implied in the master LP. In this case, we call slots $1, ..., h$ active, which tells us that K must satisfy $K \geq h$. We refer to this rule as *slot-branching*.

Second, even if a slot is active, it may carry a fractional amount of flow in the LP-relaxed master problem solution. This type of fractionality can be addressed by identifying a slot $h \in \{1, ..., H\}$ where $\hat{w}_h = \sum_{p \in P} (\hat{\lambda}^0_{hp} + f_p \hat{\lambda}^1_{hp})$ is fractional. If such a slot exists, we create two branches by imposing

$$(L) : w_h = \sum_{p \in P} (\lambda^0_{hp} + f_p \lambda^1_{hp}) \leq \lfloor \hat{w}_h \rfloor \quad \text{or} \quad (R) : w_h = \sum_{p \in P} (\lambda^0_{hp} + f_p \lambda^1_{hp}) \geq \lceil \hat{w}_h \rceil. \tag{34}$$

Additionally, because in (R) branch, we have $\lceil \hat{w}_h \rceil \geq 1$, and a solution where $y_h = 0$ and $w_h \geq 1$ is infeasible, we can fix $y_k = 1$ for all $k \leq h$. We refer to this branching rule as *flow-branching*.

Finally, even if a slot is active and its total flow is integral, the flow assigned to that slot may still be split fractionally among several compatible paths. More precisely, for some slot $h \in \{1, ..., H\}$ there may exist two distinct paths $p_1, p_2 \in P$ such that $\lambda^0_{hp_1} + \lambda^1_{hp_1}, \lambda^0_{hp_2} + \lambda^1_{hp_2} > 0$. This type of fractionality can be addressed in a couple of ways. First, we have $\forall p \in P : (\lambda^0_{hp} + \lambda^1_{hp}) \in \{0, 1\}$ if and only if $x^h = \sum_{p \in P} (\lambda^0_{hp} + \lambda^1_{hp}) \delta_p \in \{0, 1\}^{|E|}$. Hence, if the flow of slot h is split among several paths, there exists at least one arc $e \in E$ such that $0 < x^h_e < 1$. We can then branch on the use of arc e by slot h, creating two subproblems with the following additional constraints:

$$(L) : x^h_e = 0, \quad \text{or} \quad (R) : x^h_e = 1. \tag{35}$$

Equivalently, in terms of the master variables, the (L) branch forbids slot h using arc e, while the (R) branch forces slot h to use arc e.

In practice, constraint (L) can be enforced by fixing $\lambda^0_{hp} = \lambda^1_{hp} = 0$ for all $p \in P$ having $\delta_{ep} = 1$. To prevent such columns from being generated again, the pricing problem for slot h must then be solved on the modified graph $G = (V, E \backslash \{e\})$. Constraint (R), on the other hand, can be enforced by fixing $\lambda^0_{hp} = \lambda^1_{hp} = 0$ for all

$p \in P$ having $\delta_{ep} = 0$. This requires the pricing problem to generate only paths that contain arc e. For a single forced arc $e = (u, v)$, such a path can be obtained by solving a shortest path problem from s to u, adding arc e, and then solving another shortest path problem from v to t. However, when several arcs have been forced along a branch of the search tree, the pricing problem becomes more involved, as the generated path must travel through all forced arcs. In the simplest case, if k forced arcs must be visited, the pricing problem decomposes into $k + 1$ shortest path computations. Thus, repeatedly forcing arcs can significantly complicate the pricing problem at deeper nodes of the branch-and-bound tree.

An alternative method to eliminate fractional path usage for a slot is to follow the branching approach proposed by Barnhart et al. [4], which was developed as a complete branching method for integer multicommodity flow problems. First, consider a slot $h \in \{1, ..., H\}$ for which there exist two distinct paths $p_1, p_2 \in P$ satisfying $\lambda^0_{hp_1} + \lambda^1_{hp_1}, \lambda^0_{hp_2} + \lambda^1_{hp_2} > 0$. Given these two paths, we identify their divergence node d by tracing both paths from the source node s until the first node at which they use different outgoing arcs. Let these arcs be $a_1 = (d, u)$ and $a_2 = (d, v)$. Denote the set of outgoing arcs from node d as $\text{out}(d) = \{(d, v) \in E\}$. We then partition $\text{out}(d)$ into two disjoint subsets, $\text{out}_1(d)$ and $\text{out}_2(d)$, such that $a_1 \in \text{out}_1(d)$ and $a_2 \in \text{out}_2(d)$. Based on this partition, we create two branches by imposing the following constraints

$$(L): \sum_{\substack{p \in P: \\ p \cap \text{out}_1(d) \neq \emptyset}} (\lambda^0_{hp} + \lambda^1_{hp}) = 0, \quad \text{or} \quad (R): \sum_{\substack{p \in P: \\ p \cap \text{out}_2(d) \neq \emptyset}} (\lambda^0_{hp} + \lambda^1_{hp}) = 0. \tag{36}$$

Thus, in the (L) branch, slot h is forbidden from using any arc in $\text{out}_1(d)$, whereas in the (R) branch, it is forbidden from using any arc in $\text{out}_2(d)$.

These constraints do not need to be added explicitly to the master problem. Instead, they can be enforced by fixing to zero all variables corresponding to paths of slot h that use an arc in the forbidden subset. That is, in branch $i \in \{1, 2\}$, we fix $\lambda^0_{hp} = \lambda^1_{hp} = 0$ for all $p \in P$ such that $p \cap \text{out}_i(d) \neq \emptyset$. To prevent such columns from being generated again, the pricing problem for slot h is solved on the graph obtained by removing the forbidden arcs $\text{out}_i(d)$ from E. We refer to this branching rule as *Barnhart-branching*.

The following proposition shows that the sequential application of *slot-branching*, *Barnhart-branching*, and *flow-branching* is a complete branching strategy in the domain of original/compact variables via maps (20). We remind that a branching strategy is complete if it is capable of removing any fractional solution in descendant nodes, while preserving all integer-feasible solutions and keeping the search tree finite.

Proposition 2.
The sequential application of slot-, Barnhart-, and flow-branching defines a complete branching strategy for formulation (21), with integrality assessed in the original variables reconstructed through (20).
Proof:
We first show that slot-, Barnhart-, or flow branching is capable of eliminating any fractional solution in the original variables.

Let $(\hat{y}_h, \hat{x}^h, \hat{w}_h, \hat{z}^h)$ be a reconstructed solution in original variables obtained via maps (20). Suppose first that $\hat{y}_h$ is fractional. Then, slot-branching is capable

of eliminating this solution value in descendant nodes by directly modifying the y_h bounds. In the case of fractional $\hat{w}_h$, the flow-branching operates similarly.

Suppose then that $\hat{x}^h$ is fractional. In particular, this fractionality can be of two types: (1) each fractional $\hat{x}^h_e$ has the same value, or (2) there exist $e_1, e_2 \in E$ such that $\hat{x}^h_{e_1}, \hat{x}^h_{e_2} > 0$, $\hat{x}^h_{e_1} \neq \hat{x}^h_{e_2}$ and $\hat{x}^h_{e_1}$ is fractional. The case (1) occurs when $\hat{x}^h = (\hat{\lambda}^0_{hp'} + \hat{\lambda}^1_{hp'})\delta_{p'}$ for some unique $p' \in P$ with $0 < (\hat{\lambda}^0_{hp'} + \hat{\lambda}^1_{hp'}) < 1$, and for the remaining $p \neq p'$ we have $(\hat{\lambda}^0_{hp} + \hat{\lambda}^1_{hp}) = 0$. This happens only if $0 < \hat{y}_h < 1$, and therefore this kind of fractionality can be excluded by applying slot-branching. The case (2) occurs exactly when $\hat{x}^h$ is a convex combination of multiple δ_p's, i.e., there exist $p_1, p_2 \in P$ having $(\hat{\lambda}^0_{hp_1} + \hat{\lambda}^1_{hp_1}), (\hat{\lambda}^0_{hp_2} + \hat{\lambda}^1_{hp_2}) > 0$. Then, one can detect the divergence node d of p_1 and p_2 and apply Barnhart-branching which excludes the current fractional $\hat{x}^h$ in descendant nodes.

Lastly, since $\hat{z}^h$ has the same structure as $\hat{x}^h$ in terms of positive arcs, the fractionality of $\hat{z}^h$ can be either of type (1) or (2), but where $\hat{x}^h_e$ is replaced with $\hat{z}^h_e$. For case (1), flow-branching on $\hat{w}_h$ eliminates the current fractional $\hat{z}^h$. If $\hat{z}^h$ is fractional according to (2), then $\hat{x}^h$ is also fractional in a similar way, and both can be excluded in descendant nodes by applying Barnhart branching on $\hat{x}^h$.

Additionally, slot branching, flow branching, and Barnhart branching preserve all integer-feasible solutions of the current node. This is trivial for slot- and flow-branching. For Barnhart branching, each integer-feasible $s-t$ path must use at most one incoming or outgoing arc per node, and therefore it cannot simultaneously use one arc from $out_1(d)$ and another from $out_2(d)$. Hence, it also preserves each integer-feasible $s-t$ path in the current node.

Finally, repeated application of slot-Barnhart-flow branching cannot continue indefinitely because every separate branching method acts on a finite collection of objects. Because the number of slots H is finite, slot-branching can fix each y_h to either 0 or 1 only finitely many times. Because G is assumed to be a finite DAG, the set of $s-t$ paths P is finite, and Barnhart-branching can produce only a finite number of $out(d)$ partitions. Because for each slot h, the range of feasible flow ranges is bounded by $[0, M = \max_{e \in E} f_e]$, the flow branching can produce only a finite amount of intervals $[l, u]$, with integer endpoints $l \geq 0$ and $u \leq M$. Therefore, every root-to-leaf path in the branch-and-bound tree is finite, and because every rule can create only two new branches, the overall branch-and-bound tree is finite. □

By following the above branching strategy, slot-branching produces the bounds $K_{\min}$ and $K_{\max}$ yielding the slots $1 \leq h \leq K_{\min}$ active and slots $K_{\max} < h \leq H$ inactive. Let L_h and U_h denote the flow bounds produced by the flow-branching rules. Furthermore, at each node of the branch-and-bound tree, each slot h is associated with a set of forbidden arcs $F_h \subseteq E$, induced by Barnhart branching. These forbidden arcs define the set of feasible paths for slot h as $P_h = \{p \in P : p \cap F_h = \varnothing\}$. The master

problem at an arbitrary node then reads as

$$\min \quad \sum_{h=1}^{H} \sum_{p \in P_h} (\lambda^0_{hp} + \lambda^1_{hp}) \tag{37.1}$$

$$\text{s.t.} \quad \sum_{h=1}^{H} \sum_{p \in P_h} (\lambda^0_{hp} + f_p \lambda^1_{hp}) \delta_{ep} = f_e \qquad \forall e \in E \quad (\alpha) \tag{37.2}$$

$$\sum_{p \in P_h} (\lambda^0_{hp} + \lambda^1_{hp}) \leq 1 \qquad \forall h \in \{1, ..., H\} \quad (\beta) \tag{37.3}$$

$$\sum_{p \in P_h} (\lambda^0_{(h+1)p} + f_p \lambda^1_{(h+1)p}) - (\lambda_{hp} + f_p \lambda^1_{hp}) \leq 0 \qquad \forall h \in \{1, ..., H-1\} \quad (\gamma) \tag{37.4}$$

$$\sum_{p \in P_h} (\lambda^0_{hp} + \lambda^1_{hp}) = 1 \qquad \forall h \leq K_{\min} \quad (\beta^1) \tag{37.5}$$

$$\lambda^0_{hp} = \lambda^1_{hp} = 0 \qquad \forall h > K_{\max}, \ \forall p \in P_h \tag{37.6}$$

$$\sum_{p \in P_h} (\lambda^0_{hp} + f_p \lambda^1_{hp}) \geq L_h \qquad \forall h \in \{1, ..., H\} \quad (\eta^0) \tag{37.7}$$

$$\sum_{p \in P_h} (\lambda^0_{hp} + f_p \lambda^1_{hp}) \leq U_h \qquad \forall h \in \{1, ..., H\} \quad (\eta^1) \tag{37.8}$$

$$\lambda^0_{hp} \geq 0 \qquad \forall h \in \{1, ..., H\}, \forall p \in P_h \tag{37.9}$$

$$\lambda^1_{hp} \geq 0 \qquad \forall h \in \{1, ..., H\}, \ \forall p \in P_h \tag{37.10}$$

For slots not yet affected by the flow-branching, we define $L_h = 0$ and $U_h = \infty$.

With this master formulation, columns are no longer generated for slots $h > K_{\max}$ that have been forced inactive. Thus, for $h \in \{1, ..., K_{\max}\}$, the reduced costs of the variables λ^0_{hp} and λ^1_{hp} are

$$\begin{aligned}
\text{rc}_0(h,p) =& 1 - \beta_h - \sum_{e \in E} \delta_{ep} \alpha_e - 1(h \geq 2)\gamma_{h-1} + 1(h \leq H-1)\gamma_h \\
& - 1(h \leq K_{\min})\beta^1_h - \eta^0_h - \eta^1_h \\
\text{rc}_1(h,p) =& 1 - \beta_h - f_p \sum_{e \in E} \delta_{ep} \alpha_e - f_p 1(h \geq 2)\gamma_{h-1} + f_p 1(h \leq H-1)\gamma_h \\
& - 1(h \leq K_{\min})\beta^1_h - f_p \eta^0_h - f_p \eta^1_h
\end{aligned}$$

respectively. The pricing problem for λ^0_{hp} remains a shortest path problem. The pricing problem for λ^1_{hp} also remains an exact-capacity shortest path problem, as in (32), but is restricted to paths that do not use arcs in F_h. Consequently, the pricing problem can still be solved by Algorithm 2 with a dual solution input $(\alpha, \beta, \gamma, \beta^1, \eta^0, \eta^1)$ and a modified network $G_h = (V, E_h = E \backslash F_h, f_h = \{f_e\}_{e \in E_h})$. Therefore, the only additional changes are shifts in the reduced costs induced by the dual variables of the branching constraints.

3.2.1 A supplementary branching strategy

The main purpose of Barnhart branching is to remove a fractional path-combination solution while keeping pricing tractable. However, it has been reported that Barnhart

branching alone suffers from slow convergence and weak dual bound improvement [4, 43]. Since Barnhart branching separates paths according to one divergence node in one slot, its strength depends heavily on the chosen divergence node and the partition of the outgoing arcs. If the partition is unbalanced, or if many alternative paths remain after forbidding one side of the partition, the child relaxations may remain close to the parent relaxation. Furthermore, if there are multiple split and merge regions in the graph, a single divergence branching decision may only resolve one local routing ambiguity. The LP relaxation can then replace the forbidden paths by other paths that differ at later divergence nodes, so many successive Barnhart branches may be required before the slot path becomes integral [43].

Truffot and Duhamel [43] proposed a supplementary branching rule based on the number of path slots using a specific arc. We refer to this rule as the *joint arc-use branching*. Given a fractional master problem solution in terms of original/compact variables via maps (20), and an arc $e \in E$, we first define a set of slots whose path representations use e either fractionally or integrally

$$\mathcal{H}_e = \{h : x_e^h > 0, h = 1, \ldots, H\}.$$

Consider then some subset $S_e \subset \mathcal{H}_e$. In any integer solution, either all slots in S_e use arc e integrally, or at least one slot in S_e does not use e, yielding following branching

$$(L) : \sum_{h \in S_e} x_e^h = |S_e|, \quad \text{or} \tag{38}$$

$$(R) : \sum_{h \in S_e} x_e^h \leq |S_e| - 1. \tag{39}$$

In particular, the branch (L) implies that all λ-variables associated with slots in S_e and paths not containing e must be equal to zero. In the context of column generation, this means that the pricing problem should generate only paths that use arc e. Solving the pricing problem over the unmodified domain P_h would still be valid because any generated variables corresponding to paths that do not contain e would be implicitly fixed to zero. Nevertheless, this could lead to many unnecessary pricing iterations. As (L) forces every slot $h \in S_e$ to use arc e, the total flow carried by these slots must pass through e and therefore cannot exceed the arc weight f_e. Therefore, we can avoid modifying the pricing domain by replacing the exact branch (L) with the implied inequality

$$\sum_{h \in S_e} w_h \leq f_e. \tag{40}$$

Although the resulting branch is weaker than the exact condition (L), it is still stronger than the restricted form of (21.2), since it removes the incidence factor δ_{ep}. Therefore, this branch has potential to yield a stronger dual bound. Branch (R), on the other hand, is equivalent to forcing at least one slot $h \in S_e$ to not route anything on e. Hence, it can be implemented by updating the corresponding forbidden arc set to contain e.

It is noteworthy to mention that if we replace the Barnhart branching by the joint arc-use branching, the resulting slot–joint arc-use–flow branching pipeline is not

complete. Therefore, a complete path-structure branching rule, such as Barnhart branching, is still needed as a fallback. However, joint arc-use branching is still a valid branching strategy if, for the current master solution, there exists e and S_e satisfying

$$\sum_{h \in S_e} w_h > f_e \quad \text{and} \quad \sum_{h \in S_e} x_e^h > |S_e| - 1. \tag{41}$$

4 Numerical experiments

We evaluate the proposed branch-and-price algorithm using benchmark data and compare its performance with that of the compact formulation (4). We implement two branch-and-price algorithms: one following the slot–Barnhart–flow pipeline, and another following the slot–joint arc-use–Barnhart–flow pipeline.

The compact formulation and both branch-and-price algorithms are implemented using PySCIPOpt [29], which is a Python interface for SCIP (Solving Constraint Integer Programs) [23]. SCIP implements a generic branch-and-bound framework, where the user can implement the desired behavior via custom plugins. In particular, we implemented custom column generation and branching plugins, while the management of the search tree was left for SCIP. During column generation, the node-level LP-relaxations are solved using SoPlex [17, 18], which is the default solver of SCIP.

In numerical experiments, we use randomly selected subsets of the four benchmark datasets[1] introduced by Shao and Kingsford for the minimum path flow decomposition problem [39]. One dataset contains 1000 human RNA-sequencing samples from Sequence Read Archive[2] where transcripts (paths) and their abundances (weights) are estimated using Salmon [34]. A flow network is constructed by taking a union of the nodes and arcs present in the path data, and the weight for each arc is obtained by aggregating the flows of paths that go through it. Other three transcript datasets are obtained by simulation using Flux-Simulator [19] for human, mouse, and zebrafish species, where the flow networks are constructed in a similar manner. Figure 4 shows the distributions of optimal decomposition sizes and arc counts for these datasets.

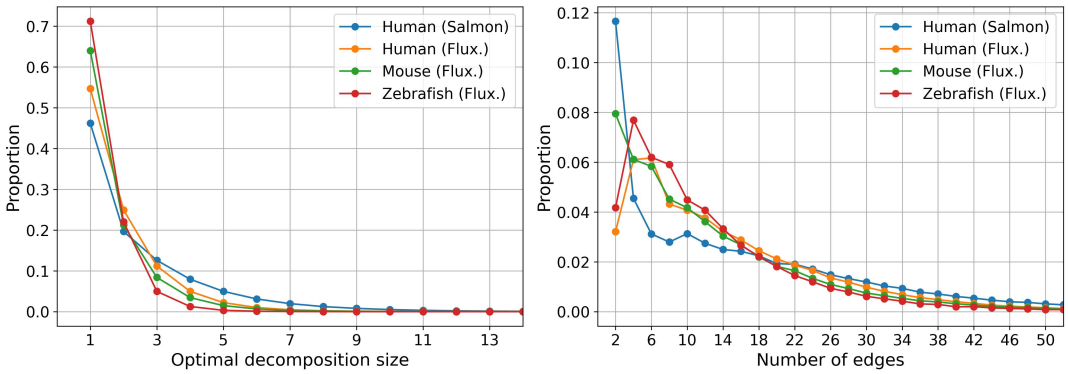


Figure 4: Proportions of ground truth minimum decomposition sizes and edge counts across all benchmark instances of Shao and Kingsford [39].

4.1 Node preprocessing and branching candidate selection

Before presenting the numerical results, we briefly describe the node-processing steps performed before branching and the heuristic rules used to select branching candidates.

[1] Full dataset is available at https://zenodo.org/record/1460998

[2] https://ncbi.nlm.nih.gov/sra/docs/

After solving the master problem, we check if its variable bounds can be strengthened, and whether these updates yield the future descendants of the current node infeasible. Let Z_{LP} be the current optimal master objective value, $K_{\min}$ and $K_{\max}$ current lower and upper bounds for the decomposition size, and Z_{inc} the best incumbent objective value. Because the decomposition size is integer-valued, we can strengthen its lower bound to $K_{\min}^{\text{new}} = \max\{K_{\min}, \lceil Z_{LP} \rceil\}$. Similarly, because we are only interested in finding strictly improving solutions, the upper bound can be updated to $K_{\max}^{\text{new}} = \min\{K_{\max}, Z_{\text{inc}} - 1\}$. If these updates do not change the current bounds, we proceed to branching. If $K_{\min}^{\text{new}} > K_{\max}^{\text{new}}$ the node is pruned because the resulting subtree could not contain an improving integer-feasible solution. Otherwise, we update the bounds of slot and flow variables affected by these modifications. All slots $h \leq K_{\min}^{\text{new}}$ are forced active, and their respective flow lower bounds are updated to $L_h^{\text{new}} = \max\{L_h, 1\}$. Accordingly, all slots $h > K_{\max}^{\text{new}}$ are forced inactive, and their flow upper bounds U_h are fixed to zero. If the new bounds yield $L_h^{\text{new}} > U_h^{\text{new}}$ for any slot, the node is pruned. Otherwise, the master problem is reoptimized.

For slot-branching, among the free slots $h \in \{K_{\min}+1, \ldots, K_{\max}\}$ having $0 < y_h < 1$, we select the one closest to the midpoint $m = \frac{K_{\min}+K_{\max}+1}{2}$. In the resulting children, this will bound the optimal decomposition size Z^* by $Z^* > m$ or $Z^* \leq m$, hence resembling a binary search over the possible decomposition sizes.

For Barnhart branching, we follow the strategy proposed in the original paper [4]. Among the slots $h \in \{1, \ldots, K_{\max}\}$ having at least two paths satisfying $\lambda_{hp}^0 + \lambda_{hp}^1 > 0$, we select the one having the largest flow value w_h. Given such an h, we identify the two paths with the greatest values of $\lambda_{hp}^0 + \lambda_{hp}^1$. When the divergence node d and the divergence arcs a_1 and a_2 are identified, the partition $\text{out}(d) = \text{out}_1(d) \cup \text{out}_2(d)$ is constructed so that the sizes of $\text{out}_1(d)$ and $\text{out}_2(d)$ are roughly equal.

For flow branching, we select the slot h where the fractional w_h has the largest value.

For joint arc-use branching, a branching candidate consists of a pair (e, S_e) satisfying (41). To select such a pair, we iterate over arcs $e \in E$, construct the corresponding $\mathcal{H}_e$, and order its slots in a decreasing lexicographic order based on pairs (x_e^h, w_h). We then consider each prefix on this ordering as a candidate S_e, and compute

$$v_L(e, S_e) = \sum_{h \in S_e} w_h - f_e \quad \text{and} \quad v_R(e, S_e) = \sum_{h \in S_e} x_e^h - (|S| - 1).$$

which measure how much the pair (e, S_e) violates (40) or (39), respectively. If either score is nonpositive, the corresponding candidate is discarded. We then select the pair maximizing

$$\min\{\frac{v_L(e, S_e)}{f_e}, v_R(e, S_e)\}.$$

Taking a minimum prefers candidates for which both resulting branches cut out the current fractional solution by a meaningful amount. Normalization by f_e is included to make the violation scores more comparable.

4.2 Results

As mentioned at the beginning of the section, in our tests we compare three different algorithms. The first algorithm, referred to as *Comp.*, solves the compact ILP formulation (4) using branch-and-bound. The other two algorithms, referred to as *Ext.1* and *Ext.2*, solve the extensive ILP formulation (21) with additional symmetry-breaking constraints via branch-and-price, that is, the corresponding master problem is (27). Furthermore, the branch-and-price procedure of *Ext.1* follows the slot–Barnhart–flow pipeline, while *Ext.2* follows the slot–joint arc-use–Barnhart–flow pipeline.

As our first experiment, we compare the total and average running times of each of the proposed algorithms over a small randomly selected sample taken from each representative dataset. The results are reported in Table 1. The column *Dec.size* gives the range of optimal decomposition sizes, and *#instances* gives the corresponding sample size. For each algorithm, we report three different statistics. Column *#solved* gives the number of samples solved to optimality within the time limit of 300 seconds. Column *Tot.all* gives the total time, in seconds, required to solve all instances. This value also include the running time of the instances that did not finish within the time limit. Column *Avg.com* gives the average solution time over instances that were solved to optimality by all three algorithms.

			Comp.			Ext.1			Ext.2		
	Dec.size	#instances	#solved	Tot.all	Avg.com	#solved	Tot.all	Avg.com.	#solved	Tot.all	Avg.com
Human (Salmon)	2–6	50	**50**	90.26	1.81	**50**	10.46	0.21	**50**	**4.53**	**0.09**
	7–10	39	38	1348.29	24.34	35	2071.95	24.91	**39**	**808.32**	**14.85**
	11–15	19	10	3276.40	38.56	10	3511.64	60.45	**12**	**2702.75**	**34.98**
Human (Flux.)	2–6	50	**50**	109.82	2.20	**50**	29.84	0.60	**50**	**13.92**	**0.28**
	7–10	38	36	1748.00	26.10	32	2900.60	23.97	**38**	**404.50**	**2.81**
	11–15	15	8	2668.01	73.95	8	2409.90	39.38	**14**	**622.95**	**7.79**
Mouse (Flux.)	2–6	50	**50**	105.92	2.12	**50**	35.57	0.71	**50**	**8.04**	**0.16**
	7–10	40	**37**	2631.15	27.67	28	4768.84	41.74	35	**2236.73**	**4.76**
	11–15	22	3	6074.90	72.63	3	5920.40	189.50	**9**	**4453.54**	**0.66**
Zebrafish (Flux.)	2–6	50	**50**	94.97	1.90	**50**	**8.35**	**0.17**	**50**	9.23	0.18
	7–10	40	34	3185.67	29.84	24	5828.23	34.12	**36**	**1929.62**	**6.07**
	11–15	11	5	2598.01	-	1	3042.91	-	**7**	**1419.86**	-

Table 1: Running time comparison between algorithms *Comp.*, *Ext.1*, and *Ext.2* across four datasets. The best results are shown in bold.

Table 1 indicates that *Ext.2* preforms best overall among the three algorithms. For instances having the optimal decomposition size between 2 and 6, all three algorithms succeed to solve all instances within the time limit. In this category, both *Ext.1* and *Ext.2* are constantly faster than *Comp.*. *Ext.2* reach the lowest total and average solution times for Salmon, Human and Mouse datasets, while *Ext.1* is marginally faster on Zebrafish dataset.

The efficiency of *Ext.2* becomes more evident when viewing the results for medium-sized instances, where the optimal decomposition size varies between 7 and

10. It solves all Salmon and Human samples within the time limit and solves more instances in the Zerbrafish dataset. Even though *Comp.* solves more instances in the Mouse dataset, *Ext.2* still has significantly lower solution times for this group. Overall, *Ext.2* has the lowest total and common average solution times in each of the four groups. In contrast, *Ext.1* solves the smallest number of instances for each dataset in this decomposition size range. In addition, it also has the highest total and common average solution times.

For the largest instances, where the decomposition size varies between 11 and 15, *Ext.2* solves more instances than either *Comp.* or *Ext.1*, and also reaches the smallest total and common average running times for every dataset. The difference is clear in each dataset. *Ext.1* again has the weakest results. It never solves more instances than *Comp.* and it solves fewer instances than *Ext.2* in all four groups. Its total and common average running times are also longer compared to *Ext.2*, and compared to *Comp.* the results are mixed.

We also performed the same previous experiment, but on a restricted set of instances for which the extensive formulation yielded greater root dual bounds. The motivation of this experiment was to see if both *Ext.1* and *Ext.2* outperform *Comp.* when their root LP-relaxations are stronger. The results are reported in Table 2. We can see that the results are very similar to those reported in the previous table. Overall, *Ext.2* solved the largest number of instances across all datasets with the fastest times. On the other hand, *Comp.* still outperformed *Ext.1* in terms of solved problems and solution times. Therefore, we may conclude that a stronger ILP formulation alone is not enough to imply better performance in practice, but the chosen branching rules also play a major role.

		Comp.			Ext.1			Ext.2		
Dataset	#instances	#solved	Tot.all	Avg.com.	#solved	Tot.all	Avg.com.	#solved	Tot.all	Avg.com.
Human (Salmon)	22	14	3074.87	31.22	12	3632.21	50.08	**15**	**2811.28**	**28.27**
Human (Flux.)	19	17	6904.11	11.42	13	2485.06	27.60	**18**	**613.54**	**2.07**
Mouse (Flux.)	43	25	6904.11	21.59	17	8663.72	50.80	**26**	**5910.03**	**8.02**
Zebrafish (Flux.)	39	27	5095.65	27.91	17	7585.30	46.59	**32**	**2967.45**	**6.46**

Table 2: Running time comparison between algorithms *Comp.*, *Ext.1*, and *Ext.2* across four datasets. The instances were selected such that the extensive formulation yielded a stronger root dual bound. The best results are shown in bold.

As our final experiment, we compared the per-instance statistics of the three algorithms. The results are reported in Table 3, where each row corresponds to a single flow network sampled from the datasets. The column *Dec.size* gives the ground truth decomposition size of the instance. For each algorithm, *Comp.*, *Ext.1*, and *Ext.2*, we report six different statistics. The column *Sol.time* gives the time required to find an optimal solution for the given instance. *Sol.time* >300 means that the algorithm did not succeed to find a solution within the time limit. Column *Gap* reports the normalized primal-dual optimality gap, given as

$$\text{Gap} = \frac{z_{\text{primal}} - z_{\text{dual}}}{z_{\text{dual}}},$$

where z_{primal} is the objective value of the best primal feasible solution found, and it is initialized with the feasible minimum flow decomposition provided by Algorithm 1. z_{dual} is the largest dual bound found. A Smaller value of *Gap* is preferred because it indicates that the best possible solution is closer to the proven optimal value. Column *Root db* gives the root node dual bound, *#Nodes* gives the number of processed branch-and-bound nodes, and *#Vars* gives the number of variables in the problem. Lastly, columns *#S*, *#J*, *#B*, and *#F* report how many times either slot-, joint arc-use-, Barnhart-, or flow-branching was used during the solving process, respectively.

Dec. size	Alg.	Sol. time	Gap	Root db	#Nodes	#Vars	#S	#J	#B	#F
4	Comp.	0.59	0.0	4.0	68	448	-	-	-	-
	Ext.1	**0.02**	0.0	4.0	**4**	**352**	0	-	3	0
	Ext.2	0.04	0.0		11	387	0	6	0	0
5	Comp.	1.20	0.0	5.0	53	644	-	-	-	-
	Ext.1	0.10	0.0	5.0	19	173	0	-	15	0
	Ext.2	**0.08**	0.0		**15**	**131**	0	11	2	0
6	Comp.	4.09	0.0	6.0	**38**	960	-	-	-	-
	Ext.1	1.33	0.0	6.0	204	550	0	-	149	0
	Ext.2	**0.77**	0.0		75	**384**	0	58	0	0
6	Comp.	7.69	0.0	6.0	196	990	-	-	-	-
	Ext.1	**0.35**	0.0	6.0	**43**	402	0	-	38	0
	Ext.2	**0.35**	0.0		45	**328**	0	31	0	0
7	Comp.	**10.83**	0.0	6.0	2716	**1036**	-	-	-	-
	Ext.1	172.83	0.0	**7.0**	13110	4164	0	-	8986	0
	Ext.2	18.83	0.0		**2337**	1511	0	1437	4	0
7	Comp.	2.12	0.0	7.0	64	624	-	-	-	-
	Ext.1	0.58	0.0	7.0	155	441	0	-	89	0
	Ext.2	**0.19**	0.0		**42**	**234**	0	26	0	0
8	Comp.	106.84	0.0	7.0	75050	1064	-	-	-	-
	Ext.1	33.34	0.0	**8.0**	2508	2555	0	-	2225	0
	Ext.2	**2.44**	0.0		**248**	**811**	0	147	7	0
9	Comp.	>300	0.11	8.0	57433	**1632**	-	-	-	-
	Ext.1	296.26	0.0	**9.0**	9555	6216	0	-	7888	0
	Ext.2	**20.70**	0.0		**845**	2681	0	511	2	0
9	Comp.	116.24	0.0	8.0	37493	**1734**	-	-	-	-
	Ext.1	183.40	0.0	**9.0**	10152	3317	0	-	6125	0
	Ext.2	**32.58**	0.0		**1963**	1888	0	1160	5	0
10	Comp.	**14.03**	0.0	10.0	557	**1920**	-	-	-	-
	Ext.1	16.30	0.0	10.0	**446**	2497	0	-	329	0
	Ext.2	15.78	0.0		536	2108	0	279	33	0
11	Comp.	>300	1.0	11.0	41555	2508	-	-	-	-
	Ext.1	>300	1.0	11.0	9623	4928	0	-	8573	0
	Ext.2	**11.01**	0.0		**465**	**1252**	0	299	1	0
11	Comp.	45.32	0.0	11.0	5708	2596	-	-	-	-
	Ext.1	>300	0.54	11.0	4156	6731	0	-	2851	0
	Ext.2	**29.24**	0.0		**1217**	**1819**	0	692	2	0
12	Comp.	**26.63**	0.0	12.0	**74**	5280	-	-	-	-
	Ext.1	112.62	0.0	12.0	2397	3501	0	-	1542	1
	Ext.2	158.41	0.0		5102	**2614**	0	2782	91	0
13	Comp.	114.47	0.0	13.0	14477	4060	-	-	-	-
	Ext.1	190.42	0.0	13.0	5763	2418	0	-	3146	0
	Ext.2	**32.66**	0.0		**1279**	**947**	0	642	5	0
14	Comp.	106.12	0.0	14.0	15696	2368	-	-	-	-
	Ext.1	>300	0.14	14.0	7235	5385	0	-	4735	0
	Ext.2	**42.51**	0.0		**2200**	**1884**	0	1223	2	0
14	Comp.	>300	0.23	13.0	70989	**2080**	-	-	-	-
	Ext.1	>300	0.14	**14.0**	15698	4333	0	-	8848	0
	Ext.2	**126.95**	0.0		**7609**	2307	0	4119	1	0
15	Comp.	>300	0.21	14.0	118896	**1972**	-	-	-	-
	Ext.1	>300	0.21	14.0	14173	4788	0	-	9752	0
	Ext.2	**144.39**	0.0		**9837**	2042	0	5561	9	0

Table 3: Comparison of the algorithms on individual instances. The best results are shown in bold.

Table 3 provides a more detailed picture of the differences between the algorithms. Similarly as in previous experiment, we observe that *Ext.2* performs best overall by solving each instance to optimality within the time limit, and both *Ext.1* and *Ext.2* provides lower solution times on smaller instances. Furthermore, *Comp.* starts to perform better on larger instances, and even outperforms *Ext.2* on three instances with decomposition sizes 7, 10 and 12, respectively.

Because *Ext.1* and *Ext.2* are based on the same extensive formulation and differ only by their respective branching strategies, they obtain the same root dual bounds. Furthermore, the dual bounds report that the extensive formulation is never weaker than the compact formulation, and for five instances, the extensive formulations reported a strictly stronger dual bound. Moreover, out of 4 of these instances, the larger root dual bound was also associated with a lower solution time of *Ext.2* compared to *Comp.*. However, the corresponding ratio for *Ext.1* is 2/5, so the stronger root dual bound does not always lead to lower solution times. What seems to best explain the lower convergence time is the number of processed nodes. In 14 out of 17 instances, a smaller number of nodes processed also produced the shortest solution time, and *Ext.2* yielded the lowest number of processed nodes in 12 instances.

The last four columns report the number of times each individual branching rule was executed during the branch-and-price solving process. First, we observe that neither *Ext.1* nor *Ext.2* used the slot-branching rule even once in any instance. This most likely due to the node preprocessing step explained in Section 4.1, where the decomposition size bounds $K_{\min}$ and $K_{\max}$ are strengthened. This often leads to fixing certain, potentially fractional, slots active or inactive and reoptimizing the master problem before entering to the branching stage. Similarly, the flow-branching counts are mainly zero for both *Ext.1* and *Ext.2*. *Dec. size*=12 is the only instance where flow branching was used only once by *Ext.1*. Therefore, for these instances, the fixed path structure seems to be often sufficient to imply integer-valued path flows as well.

That said, the majority of the computational effort in solving the extensive models comes from finding valid $s-t$ path structures. In particular, *Ext.1* may require thousands of Barnhart-branching decisions, while *Ext.2* mostly uses the joint arc-use rule and only occasionally performs Barnhart-branching at the end of the search process. This suggests that including the supplementary joint arc-use branching rule as a part of the branch-and-price procedure significantly improves the convergence time.

5 Conclusions and future work

In this thesis, we propose a branch-and-price algorithm as an exact solution method for the minimum flow decomposition problem on directed acyclic graphs. We start by reformulating a previously proposed compact ILP formulation, which constructs each path in a decomposition using arc-level variables, into an extensive formulation by applying the Dantzig-Wolfe decomposition method. We also show that the LP-relaxation of the extensive formulation is at least as strong as the LP-relaxation of the compact formulation, and for some network instances, it can be even strictly stronger. Because the extensive formulation has an exponential number of path-level variables, it is solved using the branch-and-price algorithm, which dynamically generates objective-function improving variables during the solving process.

We derive two pricing problems for the extensive model. The first is a standard shortest path problem that can be solved in polynomial time using well-known combinatorial algorithms. The second problem is a shortest path problem variant, whose objective function is multiplied by a path-specific bottleneck value, which makes its objective non-additive over the arcs and prevents it from being solved using conventional algorithms. We present a polynomial-time dynamic programming algorithm that solves this problem by explicitly enumerating all possible path-bottleneck values and solving the exact capacity shortest path problem for each of them.

To obtain integer-feasible solutions, we propose a complete branching strategy that does not complicate the column generation phase. This strategy consists of three separate branching rules, which we refer to as slot-, Barnhart-, and flow branching, that together remove the fractional solutions on levels of slot-activation, path choice, and slot-flow, respectively. These branching decisions are imposed in the domain of original/compact variables rather than the master problem domain and they could be included in the master problem via explicit branching rows, or alternatively into the pricing problem by removing arcs from the original network. This preserves the pricing problem structure during the entire solving process. We also consider a supplementary joint arc-use branching rule that is used to support the Barnhart branching rule. Instead of resolving routing ambiguity on a single slot and at a single divergence node, this rule considers the simultaneous use of an arc in several slots.

In our experiments, we compare three different approaches for solving the MFD problem. The first one solves the compact ILP formulation using branch-and-bound, while the other two solve the extensive ILP formulation via branch-and-price, with and without the joint arc-use branching rule. Specifically, we observe that the extensive formulation with the joint arc-use branching rule performs best in almost all tested MFD problem instances. The extensive formulation without the joint arc-use branching rule is also competitive on small instances, but it reports longer solution times than both alternative methods on medium and large instances. We also notice that the LP-relaxation of the extensive formulation often provides stronger root dual bounds compared to the compact formulation, but they do not necessarily imply faster solution times. Therefore, a stronger ILP formulation alone is not always enough to provide faster solution times in practice. Instead, faster solution times are almost always associated with a smaller number of processed nodes, therefore emphasizing the role

of efficient branching rules. In particular, the extensive formulation with the joint arc-use branching rule often produces smaller search trees compared to the sole use of the Barnhart branching rule.

There are several potential directions for future research. For the extensive formulation and branching methods presented in this thesis, one could, for example, experiment with alternative branching orders other than the ones proposed in this thesis. The branching rule selection could also be automated by estimating their respective pseudocosts or relying on strong branching methods [12], which explicitly estimate the potential gain of applying a specific branching rule. On the other hand, because the compact formulation tends to solve more instances and provide shorter solution times compared to the Barnhart-only branching strategy, it might be worthwhile to check how the conventional x-variable branching (35) performs in the branch-and-price framework.

Further research could also focus on developing a branch-and-price algorithm for the aggregated extensive formulation (5). Because this formulation simultaneously has a smaller solution space and is also completely unaffected by the symmetry problems present in the disaggregated formulation, it may be a promising candidate. However, the branching rules presented in this thesis are not directly applicable to this formulation. An interesting area for further research would then be to study whether artificial disaggregation methods, such as the integrality test method [48], could be used to restore the slot-level master variables. Alternatively, one could experiment with branching rules defined in the aggregated variable space, such as the subpath-branching rule proposed in [15].

Finally, one could consider solving different MFD problem variants using branch-and-price. One simple variant is to allow the flow network to contain directed cycles. In this case, if the decomposed paths must remain elementary, the resulting shortest path pricing problems would generally no longer be solvable in polynomial time [1]. Another MFD variants that could benefit from branch-and-price may include the MFD with subpath constraints, and minimum inexact flow decomposition [14].

References

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] J. Baaijens, L. Stougie, and A. Schönhuth. "Strain-aware assembly of genomes from mixed samples using flow variation graphs". In: *Research in Computational Molecular Biology*. Springer International Publishing, 2020, pp. 221–222.

[3] E. Balas. "Projection, Lifting and Extended Formulation in Integer and Combinatorial Optimization". In: *Annals of Operations Research* 140.1 (2005), pp. 125–161.

[4] C. Barnhart, C. Hane, and P. Vance. "Using Branch-and-Price-and-Cut to Solve Origin-Destination Integer Multicommodity Problems". In: *Operations Research* 48.2 (2000), pp. 318–326.

[5] C. Barnhart et al. "Branch-and-Price: Column Generation for Solving Huge Integer Programs". In: *Operations Research* 46.3 (1998), pp. 316–329.

[6] E. Bernard et al. "Efficient RNA isoform identification and quantification from RNA-Seq data with network flows". In: *Bioinformatics* 30.17 (2014), pp. 2447–2455.

[7] R. Choen et al. "On the Effect of Forwarding Table Size on SDN Network Utilization". In: 2014, pp. 1734–1742.

[8] G. Dantzig and P. Wolfe. "Decomposition Principle for Linear Programs". In: *Operations Research* 8.1 (1960), pp. 101–111.

[9] M. Dell'Amico, F. Furini, and M. Iori. "A Branch-and-Price Algorithm for the Temporal Bin Packing Problem". In: *Computers & Operations Research* 114 (2020), p. 104825.

[10] J. Desrosiers et al. *BRANCH-AND-PRICE*. Les Cahiers du GERAD, 2024.

[11] J. Desrosiers and M. E. Lübbecke. "A Primer in Column Generation". In: *Column Generation*. Springer, 2005, pp. 1–32.

[12] J. Desrosiers and M. E. Lübbecke. "Branch-Price-and-Cut Algorithms". In: *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Inc., 2011.

[13] F. H. C. Dias and A. I. Tomescu. "Accurate Flow Decomposition via Robust Integer Linear Programming". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 21.6 (2024), pp. 1955–1964.

[14] F. H. C. Dias et al. "Fast, Flexible, and Exact Minimum Flow Decompositions via ILP". In: *Research in Computational Molecular Biology*. Springer, 2022, pp. 230–245.

[15] M. Gamst et al. "Two- and three-index formulations of the minimum cost multicommodity k-splittable flow problem". In: *European Journal of Operational Research* 202.1 (2010), pp. 82–89.

[16] T. Gatter and P. F. Stadler. “Ryūtō: network-flow based transcriptome reconstruction”. In: *BMC Bioinformatics* 20.1 (2019), pp. 190–204.

[17] A. Gleixner and D. Steffy. “Linear Programming using Limited-Precision Oracles”. In: *Mathematical Programming* 183.1-2 (2020), pp. 525–554.

[18] A. Gleixner, D. Steffy, and K. Wolter. “Iterative Refinement for Linear Programming”. In: *INFORMS Journal on Computing* 28.3 (2016), pp. 449–464.

[19] T. Griebel et al. “Modelling and simulating generic RNA-seq experiments with the flux simulator”. In: *Nucleic Acids Research* 40.20 (2012), pp. 10073–10083.

[20] A. Grigorjew et al. “Accelerating ILP Solvers for Minimum Flow Decompositions Through Search Space and Dimensionality Reductions”. In: *22nd International Symposium on Experimental Algorithms (SEA 2024)*. Vol. 301. Leibniz International Proceedings in Informatics (LIPIcs). 2024, 14:1–14:19.

[21] *Gurobi Optimizer Reference Manual*. Version 13.0 (online documentation). Gurobi Optimization, LLC. 2024. URL: https://docs.gurobi.com/projects/optimizer/en/current/index.html.

[22] T. Hartman et al. “How to Split a Flow?” In: (2012), pp. 828–836.

[23] C. Hojny et al. *The SCIP Optimization Suite 10.0*. Technical Report. Optimization Online, 2025. URL: https://optimization-online.org/2025/11/the-scip-optimization-suite-10-0/.

[24] C. Hong et al. “Achieving high utilization with software-driven WAN”. In: *ACM SIGCOMM 2013 Conference, SIGCOMM 2013, Hong Kong, August 12-16, 2013*. ACM, 2013, pp. 15–26.

[25] *IBM ILOG CPLEX Optimization Studio: User’s Manual for CPLEX*. Version 22.1.2 (online documentation). IBM. 2019. URL: https://www.ibm.com/docs/en/icos/22.1.2?topic=optimizers-users-manual-cplex.

[26] M. İbrahim, Ş. İ. Birbil, and K. Bülbül. “Simultaneous Column-and-Row Generation for Large-Scale Linear Programs with Column-Dependent-Rows”. English. In: *Mathematical Programming* 142.1–2 (2013), pp. 47–82.

[27] S. Khan et al. “Safety and Completeness in Flow Decompositions for RNA Assembly”. In: *Research in Computational Molecular Biology*. Springer, 2022, pp. 177–192.

[28] K. Kloster et al. “A practical fpt algorithm for Flow Decomposition and transcript assembly”. In: *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2018, pp. 75–86.

[29] S. Maher et al. “PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite”. In: *Mathematical Software – ICMS 2016*. Springer International Publishing, 2016, pp. 301–307.

[30] S. Martin et al. “Constrained shortest path tour problem: Branch-and-Price algorithm”. In: *Computers and Operations Research* 144 (2022), p. 105819.

[31] B. M. Mumey et al. “Parity Balancing Path Flow Decomposition and Routing”. In: *2015 IEEE Globecom Workshops (GC Wkshps)* (2015), pp. 1–6.

[32] J. P. Ohst. “On the Construction of Optimal Paths from Flows and the Analysis of Evacuation Scenarios”. PhD thesis. University of Koblenz-Landau, 2015.

[33] N. Olsen, N. Kliewer, and L. Wolbeck. “A study on flow decomposition methods for scheduling of electric buses in public transport based on aggregated time–space network models”. In: *Central European Journal of Operations Research* 30.3 (2022), pp. 883–919.

[34] R. Patro, G. Duggal, and C. Kingsford. *Salmon: Accurate, Versatile and Ultrafast Quantification from RNA-seq Data using Lightweight-Alignment*. 2015.

[35] M. E. Pfetsch and T. Rehn. “A computational comparison of symmetry handling methods for mixed integer programs”. In: *Mathematical Programming Computation* 11.1 (2019), pp. 37–93.

[36] D. M. Ryan and B. A. Foster. “An Integer Programming Approach to Scheduling”. In: *Computer Scheduling of Public Transport: Urban Passenger Vehicle and Crew Scheduling*. North-Holland, 1981, pp. 269–280.

[37] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[38] M. Shao and C. Kingsford. “Efficient Heuristic for Decomposing a Flow with Minimum Number of Paths”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16.2 (2019), pp. 658–670.

[39] M. Shao and C. Kingsford. “Theory and A Heuristic for the Minimum Path Flow Decomposition Problem”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16.2 (Mar. 2019), pp. 658–670.

[40] R. Spliet. “A Simple Perspective on Simultaneous Column and Row Generation”. In: *Operations Research Forum* 5.3 (July 2024), p. 69.

[41] L. Taccari. “Integer programming formulations for the elementary shortest path problem”. In: *European Journal of Operational Research* 252.1 (2016), pp. 122–130.

[42] A. I. Tomescu et al. “Explaining a Weighted DAG with Few Paths for Solving Genome-Guided Multi-assembly”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 12.6 (2015), pp. 1345–1354.

[43] J. Truffot and C. Duhamel. “A Branch and Price algorithm for the k-splittable maximum flow problem”. In: *Discrete Optimization* 5.3 (2008), pp. 629–646.

[44] P. H. Vance et al. *A Heuristic Branch-and-Price Approach for the Airline Crew Pairing Problem*. Technical Report TLI/LEC-97-06. Georgia Institute of Technology, 1997.

[45] F. Vanderbeck. “On Dantzig-Wolfe Decomposition in Integer Programming and Ways to Perform Branching in a Branch-and-Price Algorithm”. In: *Operations Research* 48.1 (2000), pp. 111–128.

[46] F. Vanderbeck. “Branching in branch-and-price: a generic scheme”. In: *Mathematical Programming* 130.2 (2011), pp. 249–294.

[47] F. Vanderbeck and L. A. Wolsey. “An Exact Algorithm for IP Column Generation”. In: *Operations Research Letters* 19.4 (1996), pp. 151–159.

[48] F. Vanderbeck and L. A. Wolsey. “Reformulation and Decomposition of Integer Programs”. In: *50 Years of Integer Programming 1958–2008*. Springer, 2010, pp. 431–502.

[49] B. Vatinlen et al. “Simple bounds and greedy algorithms for decomposing a flow into a minimal set of paths”. In: *European Journal of Operational Research* 185.3 (2008), pp. 1390–1401.

[50] D. Villeneuve et al. “On Compact Formulations for Integer Programs Solved by Column Generation”. In: *Annals of Operations Research* 139.1 (2005), pp. 375–388.

[51] L. Williams, G. Reynolds, and B. Mumey. “RNA Transcript Assembly Using Inexact Flows”. In: IEEE, 2019, pp. 1907–1914.