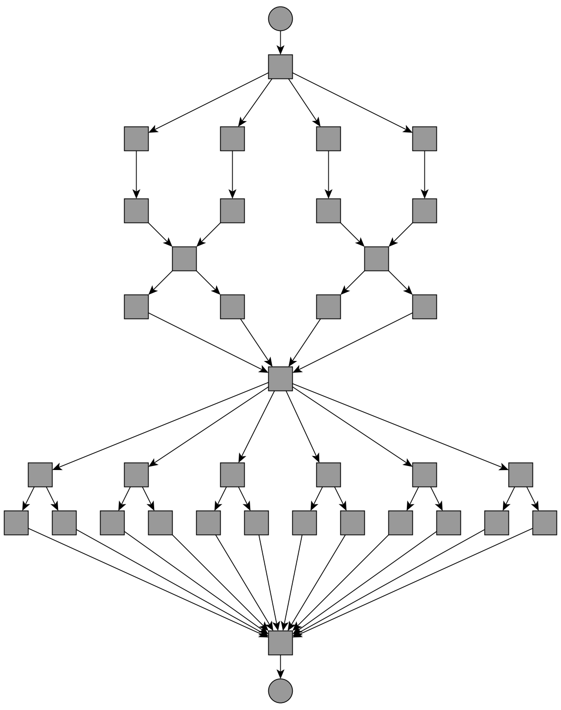
CHALMERS
UNIVERSITY OF TECHNOLOGY



Deploying processing functions on a many-core grid:

Mathematical optimization modelling and methodology

Master's thesis in Engineering Mathematics and Computational Science

NILS FREDRIKSON

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019

Deploying processing functions on a many-core grid:

Mathematical optimization modelling and methodology

NILS FREDRIKSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Research group of Mathematical Optimization
Chalmers University of Technology
Gothenburg, Sweden 2019

Deploying processing functions on a many-core grid:
Mathematical optimization modelling and methodology
NILS FREDRIKSON

Supervisors: Ann-Brith Strömberg, Department of Mathematical Sciences,
Peter Brauer, Ericsson AB
Examiner: Ann-Brith Strömberg, Department of Mathematical Sciences

Master's Thesis 2019
Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Research group of Mathematical Optimization
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A graph describing the dependencies between ordered signal processing functions. This particular processing graph is used as a test case in the experiments carried out in this work.

Typeset in LaTeX
Printed by the Department of Mathematical Sciences
Gothenburg, Sweden 2019

Deploying processing functions on a many-core grid:
Mathematical optimization modelling and methodology
NILS FREDRIKSON
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

When hardware platforms with over 1000 logical cores are used for signal processing, the optimization of latency and core utilization is key for efficient processing. To describe signal processing software, a consistent tool is that of a processing graph, where each node represents a data processing function. The problem of optimally deploying such processing functions from a graph, onto a many-core platform in the form of a grid, is considered in this thesis. A mathematical optimization model in the form of a mixed-binary linear programming model is proposed by decomposing the processing graph using a function sequence framework. A case processing graph is then deployed to two different grids by using the branch-and-bound method to solve the models to optimality. To cope with the computational complexity of branch-and-bound for large grids and processing graphs with many nodes, a column generation approach is taken. To this end, two different Dantzig-Wolfe decompositions of the model are made, each dividing the original model formulation into one master and one subproblem. In the first decomposition, the subproblem constraint matrix is block-diagonal. The second decomposition lacks this property. The column generation method is then applied to each of the two decompositions to search for approximately optimal solutions to the test cases. Column generation is then compared to branch-and-bound in terms of resulting binary solutions, objective values, and computational running time. The results indicate that the column generation can be used to search for improvements in an initial feasible solution. But the improvements were very small for the current proposed model and decompositions. To further develop the method it is suggested that constraints or costs that reduce symmetries within the model, are introduced. Compared to branch-and-bound, the running time shows that column generation has the potential to be a viable alternative method to solve the deployment problem.

Keywords: signal processing, processing graph, mixed-integer linear optimization, mathematical modelling, branch-and-bound, column generation, Dantzig-Wolfe decomposition.

Acknowledgements

I would like to thank Ann-Brith, my supervisor at Chalmers, for the many interesting and helpful discussions regarding the modelling and method development. At Ericsson, I especially want to thank Peter and Martin for the continuous support and guidance during the entire course of this project. I am very grateful for the freedom you gave me regarding the direction of my thesis work, and the possibility to shape it from the very beginning. For help with the implementation of my methods, I would like to give a special thanks to Edvin Åblad at Fraunhofer Chalmers.

Nils Fredrikson, Gothenburg, June 2019

Table of Contents

1

Introduction

Hardware platforms for signal processing have during the last decades evolved towards grids of digital signal processors (DSPs) connected by a transport network. Today, chips with 1000 cores exist and the trend suggests that this number will increase further [1]. Each DSP is designed to process data using configurable filters and algorithms. The possibility to program user-defined functions onto the hardware layout results in great flexibility and is one of the advantages of this network. This configurability also opens up for optimizing of DSP utilization and processing speed. For the application considered in this work, there is a continuous flow of data through the grid of DSPs and a given software layout will therefore be static. The minimization of latency is then especially important.

From a software perspective, a consistent description of signal processing is to use an acyclic directed graph [2], where each node represents a data processing function. The transport network structure on the chip corresponds to edges which connect the nodes. Several parallel graphs can be deployed on a chip with DSPs and interconnected through the transport network. In simpler cases this means that each node is associated with a unique DSP, but depending on the complexity of the configuration, nodes belonging to different graphs can share a DSP.

The task investigated in this thesis is to optimize the deployment of graphs with nodes onto the grid of DSPs, in order to minimize the latency of processing. This problem is often referred to as *task mapping*. Communication between cores comes with a cost, since the length of a data transport path affects said latency. Furthermore, the network transport connections between DSPs are point-to-point, meaning that when a communication path is established it cannot be shared freely with other DSPs. Thus, longer paths use more of the available connections and might severely reduce the utilization of cores.

The problem described above is so-called NP-complete, meaning that the number of feasible (i.e., possible) solutions is exponential in the number of graphs and nodes. To search for every possible solution is therefore not computationally feasible. One possible solution method is to use heuristics, such as simulated annealing [3]. This method could provide feasible solutions that yield upper bounds on the value of an optimal solution. However, to quantitatively measure the quality of a solution—as compared to an optimal solution—a concise mathematical modelling is needed. To tackle the computational complexity of the problem, an approach using so-called *column generation* [4], will be made in this work.

2

Problem Formulation and Specification

We present a detailed description of the aspects of the problem components that will be taken into account in the optimization model. Limitations regarding the signal processing functions and the processor elements on the grid network, are presented. We also introduce function sequence structures to describe the function dependencies in the processing graphs. These structures will form a basis for the formulation of decision variables and constraints, and will thus aid in the construction of the mathematical optimization model in Chapter 4. The specifics and limitations formulated in this section will make up the scope of this work.

2.1 Processing Graph Decomposition

A simple example of a processing graph is presented in Figure 2.1. The graph consists of a set of nodes, indicated by circles and squares, and a set of directed arcs, which interconnect the nodes. The square nodes represent the input and output of data (I/O), while the circles represent the processing functions. Each circular node is associated with a signal processing function, e.g., a Discrete Fourier transform (DFT), that will act on the flow of data packages sent through the platform of DSPs. We will denote these processing functions s_1,s_2, . . ., but we will not consider the actual function actions or their effect on the data. Therefore the processing graph simply describes the order of processing and the dependencies between the processing functions. The rectangular nodes representing I/O, will in this approach be treated in the same manner as the processing functions, in the sense that there is a decision to be made regarding their assignment to a DSP on the grid network. However, there is no actual processing done in these ”functions”. A restriction on the I/O functions is that they should be assigned to DSPs residing on the top/bottom rows on the grid, respectively. In Figure 2.1, node 1 has arcs to both nodes 2 and 3. The interpretation of this is that after data has been processed by the function s_1 in node 1, the resulting data should be distributed to the processing functions s_2 and s_3. In the case of processing parallelism, i.e., that $s_2 = s_3$, the data will be distributed evenly between the nodes. If $s_2 \neq s_3$ then the

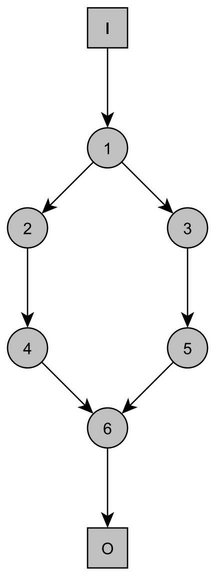


Figure 2.1: Example of a processing graph.

distribution can be weighed differently. In any case, the distribution of the resulting data between processing functions is assumed to be determined beforehand by the system and will not be taken into account in the optimization model. The relations between the nodes 4, 5, and 6 are analogous. Nodes such as 1 and 6, which have more than one directed arc to and/or from it, will be called *connection nodes*. These nodes will be important when defining *sequences* of processing functions.

We will now define the *function sequence* framework which will aid in the construction of the optimization model. A function sequence is a grouping of the nodes, and thus processing functions, in the processing graph. Each sequence consists of a number of consecutive processing functions, which follow the order as given by the processing graph. The sequences are constructed such that they always start and/or end in a connection node. Specifically, the sequences containing the I/O functions end/start in a connection node, respectively, while the intermediate sequences both start and end in connection nodes. Further, a sequence is defined by a set of consecutive nodes, with a directed arc between any two adjacent nodes. In Figure 2.2 function sequences for the processing graph in Figure 2.1 are visualized. The line patterns of the arcs between the pairs of nodes indicates the consecutive processing functions in the same function sequence. All nodes connected by arcs of identical pattern belong to one sequence. In the figure there are thus four sequences, visualized by full line (nodes I, 1), dashed line (1, 2, 4, 6), dotted line (1, 3, 5, 6), and dash-dotted line (6, O).

As illustrated in Figure 2.2, these rules for the construction of function sequences mean that any processsing function s_t associated with a connection node t will belong to multiple function sequences. To enable referencing functions in sequences and modelling, some new

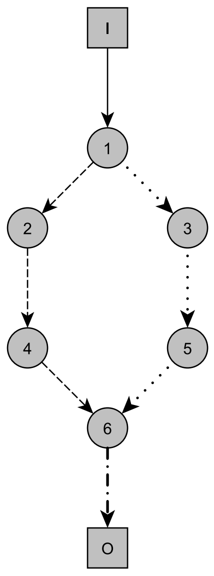


Figure 2.2: Function sequences in a processing graph.

notation is needed. We will denote the set of sequences in a graph by $\mathcal{M} := \{1, 2, \ldots, M\}$ and denote the functions in sequence m by $f_{m,n}$, $n \in \mathcal{N}_m$, where $\mathcal{N}_m := \{1, 2, \ldots, N_m\}$, denotes the ordered set of functions in sequence $m \in \mathcal{M}$. The set of connection nodes in the processing graph will be denoted by $\mathcal{T}$. This notation means that we can separate the processing functions $s_1, s_2, \ldots$, associated with the nodes in the processing graph, from the abstract functions $f_{m,n}$ in the function sequences. For example, let's say that the function sequence represented by dashed arcs in Figure 2.2 is numbered $m = 2$. This sequence contains the nodes 1, 2, 4, and 6, and they are denoted $f_{2,1}$, $f_{2,2}$, $f_{2,3}$ and $f_{2,4}$. Similarly, the sequence represented by dotted arcs, say $m = 3$, contains the functions $f_{3,1}$, $f_{3,2}$, $f_{3,3}$ and $f_{3,4}$. Denoting the sequence represented by full line arcs by $m = 1$, then the node 1 in the processing graph is represented in the sequences 1, 2, and 3, and associated with the functions $f_{1,2}$, $f_{2,1}$ and $f_{3,1}$.

It is not uncommon to have the nodes of several processing graphs mapped to a single many-core platform. This may include multiples of identical graphs or a mix of different graphs. Using the sequence structure defined above, these cases can be covered in the same manner as with a single graph and the model created in Section 4 allows for these scenarios as well. However, they are not part of the tests in this work.

2.2 DSP Grid

Each node in a processing graph, such as the one in Figure 2.1, corresponds to a computational task that should be carried out by a processor element. In this work, the overall application is signal processing, so therefore the DSPs constitute the processor elements. These DSPs are positioned on a rectangular grid and connected through a data transportation network. We denote the ordered set of rows and columns on the grid by $\mathcal{I} := \{1, 2, \ldots, I\}$ and $\mathcal{J} := \{1, 2, \ldots, J\}$, respectively. On every position $(i, j) \in \mathcal{I} \times \mathcal{J}$ is a DSP, to which there is a possibility to assign a processing function from the graph. In general, the number of processing functions mapped to a DSP depends on the computational capacity of the DSP and the required effort to complete the processing. For example, one could choose to assign a single highly computationally demanding task to a DSP, or several less demanding tasks. However, in this work we will limit the task mapping so that every processing function mapped to a DSP will demand the entire computational capacity; hence DSPs are not shared by multiple processing functions from the graph.

Each DSP has a number of connections to its horizontal and vertical neighbouring DSPs, through which data can be sent and recieved. Depending on the system architecture, the number of connections between neighbours and the directions of the connections, may vary. For example there could be two connections between vertical neighbours and four between horizontal ones, where half of the connections on each side can be used for transmitting data, and half for receiving data. In this work we consider equal numbers of connections in all directions, although the number is varied between test instances, and the directions of the connections are not pre-defined by the system. This means that the result of the mapping decides the direction of each connection, but there is a limited number of connections to choose from. Figure 2.3 visualizes a set of DSPs on a grid network with two connections between each pair of neighbours.

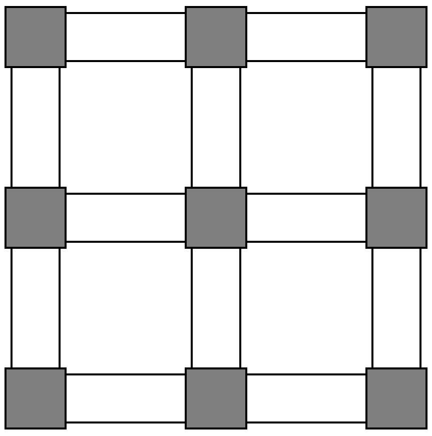


Figure 2.3: Example of a DSP grid with two connections between each neighbour.

Elaborating on the above, each position on the grid serves two purposes. The first is as a DSP location, and by that as a potential candidate for a processing assignment. The second is as a connection switch. Thus, a DSP's connections constitute part of a collective pool of network paths that can be used to transport data between any DSPs associated with any processing functions. The connections of a DSP are thereby not restricted to the transport of data associated with the processing assigned to that DSP. In Figure 2.4 the DSPs positioned at $(1, 1)$ and $(1, 3)$ have been assigned two consecutive processing functions, e.g., nodes I and 1 in the full line sequence ($m = 1$) in Figure 2.2, and there is a need to transport data from position $(1, 1)$ to position $(1, 3)$. Similarly, the DSPs positioned at $(1, 2)$ and $(2, 2)$ were assigned nodes 2 and 4, respectively, and are communicating analogously. As depicted in Figure 2.4, position $(2, 2)$ is also used to transport data between $(1, 1)$ and $(1, 3)$, without being part of their processing chain. With this logic, a position on the grid can also be used solely as a connection switch.

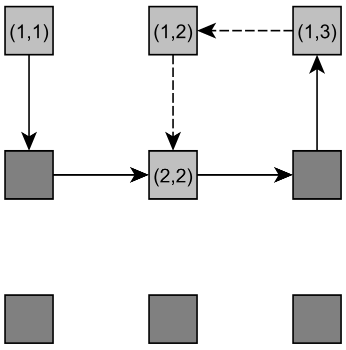


Figure 2.4: Data transport through switches.

Like the processing functions of the graph are to be assigned to a DSP on the grid, the arcs between the processing functions are to be assigned to a path made up by connection segments between the DSPs on the grid. When an arc has been assigned to a connection segment, this segment cannot be used by arcs in other chains of processing functions, and is thus occupied. This characteristic sets a significant limitation on the mapping possibilities.

2.3 Problem Statement and Limitations

Having specified the problem components we can more closely formulate the problem statement and summarize some of the limitations made above. The goal of this work is to find a mapping of processing functions from a given graph to a grid of DSPs, such that the quality of the solution can be quantitatively measured with regard to optimality fulfilment. Communication between cores comes with a cost, since the length of a data transport path affects the latency of processing. Furthermore, the network transport connections between DSPs are point-to-point, meaning that when a communication path is established it cannot be shared freely with other DSPs. Thus, longer paths use more of the available connections and might severely reduce the utilization of cores. To this end, we will create a mathematical model of the problem and, within the framework of mathematical optimization, use methods to search for optimal and/or approximately optimal solutions to this problem.

Seeing to the apparent network structure of this problem we will create a mixed-binary linear programming (MBLP) model, using binary decision variables and formulating constraints representing the characteristics described in Sections 2.1–2.2. The objective will be to minimize the number of connection segments needed for each processing step to achieve operational signal processing. The initial MBLP model can be solved using the branch-and-bound method. However, when analyzing the computational complexity of this algorithm, it will be clear that an alternative method is needed as the sizes of the problem instances grow. For the task mapping problem that could mean the mapping of additional graphs or more complex ones, larger grid of DSPs, or a combination of the two. To tackle this issue we will adapt a column generation approach. A transformation of the MLBP model into a Dantzig-Wolfe decomposition with a master and subproblem structure will be made to fit this approach. Finally, the results of the different methods will be compared with regards to optimality and computational running time.

As a final remark, we recapture the limitations mentioned in this chapter. Below follows a list of problem characteristics that will narrow the scope of this work.

- Processing functions mapped to a DSP will fully occupy it and hence no other functions can be mapped to it. That is, different processing functions cannot utilize the same processor.
- The distribution of data between nodes will be determined by the system and its graph, and not by the mapping.
- Optimization will consider the ordering of processing functions and not their actual functionality. Hence their altering of the data flow will be ignored.
- Computational effort required by individual processing functions will not be taken into account. Thus processing running times and scheduling aspects will not be considered.
- The DSPs on the grid are homogeneous. Every processor has the same capability to carry out a processing task.
- Nodes corresponding to I/O will also be treated as processing functions and will in a feasible solution be assigned to a DSP on the grid. Also, the I/O functions should be assigned to DSPs residing on the top and bottom rows on the grid, respectively.

3

Optimization Modelling and Methodology

In this chapter the necessary theory and methods on the subject of mathematical optimization is introduced. A mixed-binary linear programming (MBLP) model will be the approach for investigating the problem defined in the previous chapter, and to that end a number of topics need to be covered.

The starting point will be to go over the topics concerning the fundamental optimization techniques applied. These topics include an introduction to linear programming (LP), the simplex method and LP duality, which are central to all methods used in this work. This part will be based on chapters 3, 6, 8, 9 and 10 in [5].

Based on the works of Sierksma [6], we will then cover how logical constraints in an optimization problem are translated into equalities and inequalites using binary variables corresponding to boolean statements. These types of constraints are typical for problems with network structures, like the problem formulated in this work. Therefore some methods and examples for modelling networks are included in this section. Further, we will introduce MBLP and the branch-and-bound algorithm used to solve these problems.

To motivate our column generation approach to the optimization problem, some computational complexity aspects regarding MBLP, will be covered. This includes a discussion of the NP-completeness of the branch-and-bound algorithm. Most of this is a summary of concepts discussed by Schriejver and Sierksma in [7] and [6], respectively. Finally, following that of [4], a thorough description of the column generation method and how it can be embedded in a branch-and-bound framework using Dantzig-Wolfe decomposition, will be given. This will include how duality theory is used to calculate bounds to measure convergence of the column generation algorithm, and how to take advantage of a block-diagonal structure in the constraint matrix.

3.1 Linear Programming

A general optimization problem can be stated as the problem to find

$$\begin{aligned} \min_{x} \quad & f(x), \\ \text{subject to} \quad & x \in X. \end{aligned} \tag{3.1}$$

where $X \subseteq \mathbb{R}^n$ and $f : \mathbb{R}^n \mapsto \mathbb{R}$ is some function.

A linear programming (LP) model is a special case of an optimization model where the objective function is linear and the feasible region is defined by linear equalities and inequalites. Let's consider the case of n decision variables $x = (x_1, x_2, \ldots, x_n)^\top$. This means that in the problem (3.1), we can write the objective function $f(x)$ as a linear function $z = c^\top x$, with $c \in \mathbb{R}^n$, and define the feasible region X by a convex polyhedron

$$X = \{x \in \mathbb{R}^n \mid Ax \leq b;\ x \geq 0\}, \tag{3.2}$$

where the matrix $A \in \mathbb{R}^{m \times n}$ and the vector $b \in \mathbb{R}^m$. For the purposes of this work, it is sufficient to consider bounded polyhedra X. An example of such a polyhedron in $\mathbb{R}^2$ is shown in Figure 3.1. The extreme points of the region are also shown. These points are of great importance when solving LP problems, as will be seen in Section 3.1.1.

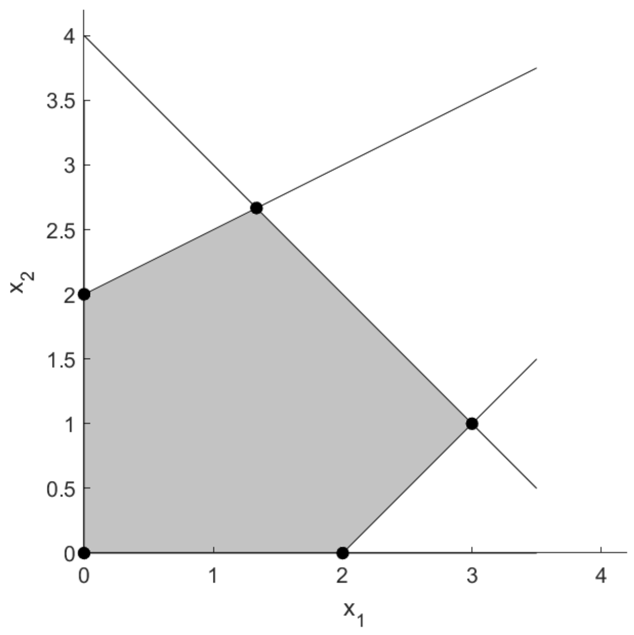


Figure 3.1: The feasible region in LP. Extreme points are represented with dots.

Definition 1 (Extreme points)**.** Let C be a convex set with $x^1, x^2 \in C$, and let $\lambda \in (0, 1)$. A point v of C is called an extreme point if the relation $v = \lambda x^1 + (1 - \lambda)x^2$ implies the equalities $v = x^1 = x^2$.

The extreme points of a bounded polyhedron can be thought of as the points defining the polyhedra, since using only convex combinations of the extreme points, every point in the polyhedron can be defined.

Continuing with the LP formulation, the complete minimization problem now reads

$$\begin{aligned} \min_{x} \quad & z = c^{\top}x, && (3.3a)\\ \text{subject to} \quad & Ax \leq b, && (3.3b)\\ & x \geq 0. && (3.3c) \end{aligned}$$

In this representation we have chosen non-negative decision variables x and an inequality of the type "$\leq$" for the constraints, with a constraint matrix A and a right-hand side b. Given variables and inequalities one can always transform a linear program into (3.3) by changing signs in the constraints corresponding to (3.3b) and/or (3.3c). Similarly, one can transform a maximization problem into a minimization problem by changing the sign of the objective function.

3.1.1 Simplex Method

One of the most common methods of solving LP models is the simplex algorithm. The algorithm relies on the geometry of linear programs, more specifically the equivalence between optimal solution candidates and extreme points of the polyhedron that defines the feasible region.

To apply the simplex method, one must first transform the LP model into a so-called *standard form*. There are versions of standard forms for both minimization and maximization problems, but for our purposes the minimizing form is more suitable:

$$\begin{aligned} \min_{x} \quad & z = c^{\top}x, && (3.4a)\\ \text{subject to} \quad & Ax = b, && (3.4b)\\ & x \geq 0. && (3.4c) \end{aligned}$$

We can transform the model (3.3) into standard form by adding slack variables $s \in \mathbb{R}^m$ to the inequality constraints $Ax \leq b$. That is, for each inequality in (3.3b) we add a variable $s_i \geq 0$ according to

$$a_i x + s_i = b_i,$$

where $a_i \in \mathbb{R}^n$ and $b_i \in \mathbb{R}$.

The next step in the simplex algorithm involves a partitioning of the problem to fit the concept of basic and non-basic variables, as well as basic feasible solutions (BFS). If for an LP model in standard form it holds that rank $A = m$, $n > m$ and $b \geq 0$, then $\tilde{x} \geq 0$ is a BFS if

1. $A\tilde{x} = b$,
2. columns of a A corresponding to non-zero components of $\tilde{x}$ are linearly independent.

We will now partition x and A to fit the above description:

$$x = \begin{pmatrix} x_B \\ x_N \end{pmatrix}, \quad A = (B, N), \tag{3.5}$$

where $x_B \in \mathbb{R}^m$, $x_N \in \mathbb{R}^{n-m}$ are called basic and non-basic variables, respectively, and $B \in \mathbb{R}^{m \times m}$, $N \in \mathbb{R}^{m \times (n-m)}$. Using this partitioning we can now write

$$Ax = Bx_B + Nx_N = b. \tag{3.6}$$

This partitioning comes naturally since rank $A = m$, yielding the possibility to solve the system $Ax = b$ with only m elements of the n-vector x corresponding to linearly independent columns of A. Hence, those m variables are found in x_B and the other $n - m$ variables, found in x_N, can be set to 0. By the same logic, the independent columns of A corresponding to the m variables in x_B are placed in the basis B and the remaining columns are placed in N. A BFS to (3.4) is now given by

$$x = \begin{pmatrix} x_B \\ x_N \end{pmatrix} = \begin{pmatrix} B^{-1}b \\ 0 \end{pmatrix}. \tag{3.7}$$

Having a BFS x and a corresponding partitioning as above, we can express x_B and the objective $c^\top x$ according to

$$x_B = B^{-1}b - B^{-1}Nx_N, \tag{3.8a}$$

$$\begin{aligned} c^\top x &= c_B^\top x_B + c_N^\top x_N \\ &= c_B^\top (B^{-1}b - B^{-1}Nx_N) + c_N^\top x_N \\ &= c_B^\top B^{-1}b + (c_N^\top - c_B^\top B^{-1}N)x_N. \end{aligned} \tag{3.8b}$$

As outlined in more detail in [5, Ch. 8–9], the idea behind the simplex method is now to use the fact that a BFS is actually equivalent to an extreme point to the feasible region defined by the constraints (3.4b)–(3.4c), and that the global optimum of the same problem is found among the extreme points. Starting at a BFS x, one uses the simplex method to move between adjacent extreme points and search for improvements in the objective.

The optimality criteria involves the *reduced costs* for the non-basic variables x_N derived in (3.8). The reduced cost $\tilde{c}_j$ for a non-basic variable x_j, $j = 1, 2, \ldots, n - m$, in x_N is defined by

$$\tilde{c}_j = (c_N^\top - c_B^\top B^{-1}N)_j, \tag{3.9}$$

corresponding to the j:th element in the vector $c_N^\top - c_B^\top B^{-1}N$. If $\tilde{c}_j < 0$ for at least one of the variables in x_N, then the column in N corresponding to the largest decrease, per unit increase in the corresponding variable x_j, in the objective will be swapped into the basis B, replacing the column corresponding to the basic variable that first reaches the value 0 when the value on x_j increases, and a new extreme point is identified. When $\tilde{c}_j \geq 0$ holds for all j, the current basis B is optimal and we have found an optimal solution x to (3.4).

3.1.2 Duality Theory

An important concept in optimization theory is that of duality and dual problems. For every optimization problem (here called the primal problem) there exists a dual problem, and the properties and solutions of the dual problem can in many cases provide information about the primal problem. The approach in this work is in many ways reliant on duality theory, more specifically linear programming duality, and the most important concepts are outlined in this subsection.

The fundamental starting point for duality theory is the relaxation theorem. A relaxation is a reformulation of a general optimization problem

$$\begin{aligned} \min_{x} \quad & f(x), \\ \text{subject to} \quad & x \in X, \end{aligned} \tag{3.10}$$

where $X \subseteq \mathbb{R}^n$ is closed and bounded with $f : X \mapsto \mathbb{R}$, into a related problem

$$\begin{aligned} \min_{x} \quad & f_R(x), \\ \text{subject to} \quad & x \in X_R, \end{aligned} \tag{3.11}$$

where $X \subseteq X_R$ and $f_R \leq f$, for $x \in X$. The relaxation theorem then formulates some basic properties of a relaxation.

Theorem 1 (Relaxation Theorem)**.** *For a relaxation as defined in* (3.11) *the following holds:*

1. *For an optimal solution $\hat{x}_R$ to* (3.11) *and an optimal solution $\hat{x}$ to* (3.10) *we have that $f_R(\hat{x}_R) \leq f(\hat{x})$.*
2. *If* (3.11) *is infeasible, then so is* (3.10)*.*
3. *If the optimal solution $\hat{x}_R$ to* (3.11) *satisfies $\hat{x}_R \in X$, and $f_R(\hat{x}_R) = f(\hat{x}_R)$, then $\hat{x}_R$ is optimal in* (3.10) *as well.*

A simple proof of Theorem 1 can be found in [5, pp. 157–158]. This result plays a key role in the branch-and-bound method formulated in Section 3.5.

Moving on, a natural relaxation of an optimization problem is the *Lagrangian relaxation.*

Consider a more specific optimization problem of the form

$$\begin{aligned}
&\min_{x} && f(x), && (3.12a)\\
&\text{subject to} && g_i(x) \leq 0, \quad i = 1, 2, \dots, m, && (3.12b)\\
& && x \in X, && (3.12c)
\end{aligned}$$

with f and X defined as above and $g_i : \mathbb{R}^n \mapsto \mathbb{R}$, $i = 1, 2, \dots, m$, being functions. The Lagrangian relaxation can now be formulated.

Definition 2 (Lagrangian Relaxation)**.** Consider the function $L : \mathbb{R}^{n\times m} \mapsto \mathbb{R}$, defined by $L(x, \mu) := f(x) + \sum_{i=1}^{m} \mu_i g_i(x)$, and let $0 \leq \mu_i$, for $i = 1, 2, \dots, m$. A Lagrangian relaxation of (3.12) with respect to the constraints (3.12b), is given by

$$\begin{aligned}
q(\mu) := &\min_{x} \quad L(x, \mu),\\
&\text{subject to} \quad x \in X.
\end{aligned} \tag{3.13}$$

The function L is referred to as the Lagrange function. Since $0 \leq \mu_i$ and $g_i(x) \leq 0$ for all $i = 1, 2, \dots, m$ by definition, the Lagrangian relaxation (3.13) of (3.12) is a relaxation in terms of (3.11), above.

Taking a step closer to duality and dual problems, the Lagrangian dual problem can now be formulated as

$$\max_{\mu \geq 0} \quad q(\mu) := \min_{x \in X} L(x, \mu), \tag{3.14}$$

where the function $q : \mathbb{R}^m \mapsto \mathbb{R}$. Thus, the dual problem of a minimization problem is a maximization problem, and for every constraint $i = 1, 2, \dots, m$ in the primal problem, a decision variable μ_i is included in the dual problem. The dual variables μ_i determines a penalty of violating the relaxed constraints $g_i \leq 0$. In (3.13) feasibility still holds if these constraints are violated, but it would be costly.

As stated in the introduction of this section, the dual problem provides important information about the original primal problem. Next is a very important result relating the dual to the primal problem.

Theorem 2 (Weak Duality)**.** *For any x and μ feasible in* (3.12) *and* (3.14)*, respectively, we have that*

$$q(\mu) \leq f(x).$$

Proof. Let $\mu \in \mathbb{R}^m$ and $x \in X \subseteq \mathbb{R}^n$, with $0 \leq \mu$ and $g(x) \leq 0$, where $g : \mathbb{R}^n \mapsto \mathbb{R}^m$. This gives

$$q(\mu) = \min_{v \in X} L(v, \mu) \leq f(x) + \mu^\top g(x) \leq f(x),$$

where the equality holds by the definition of the function q in (3.14), the first inequality holds by the definition of the function L, and the second inequality holds since $\mu \geq 0$ and $g(x) \leq 0$. Especially for optimal solutions $\hat{x}$ to (3.12) and $\hat{\mu}$ to (3.14), we have

$$q(\hat{\mu}) = \max_{\mu \geq 0} q(\mu) \leq \min_{x \in X : g(x) \leq 0} f(x) = f(\hat{x}). \tag{3.15}$$

□

Thus the dual problem yields a lower bound on the optimal value of any primal problem. This result is referred to as *weak duality*, and is a general result in optimization. However, for some (i.e., convex) problems the inequality in (3.15) can be replaced by equality, giving so-called *strong duality.* This is the case for linear optimization problems.

So far a general approach to duality theory has been presented. For linear programming the conventions introduced in Section 3.1.1 are used to formulate the dual problem, and the weak and strong duality theorems. Consider (3.4), an LP problem in standard form. Following the rules stated above to form the dual problem formulation, the dual of (3.4) is

$$\begin{aligned} \max_{y \in \mathbb{R}^m} \quad & q = b^\top y, \\ \text{subject to} \quad & A^\top y \leq c. \end{aligned} \tag{3.16}$$

The weak duality theorem can now be stated for LP problems.

Theorem 3 (Weak Duality for Linear Programming)**.** *If x is feasible in* (3.4) *and y is feasible in* (3.16)*, then the inequality $b^\top y \leq c^\top x$ holds.*

Proof. Using the relations $A^\top y \leq c$, $0 \leq x$, and $Ax = b$ we have that

$$b^\top y = y^\top b = y^\top Ax = (A^\top y)^\top x \leq c^\top x,$$

and the claim is proved. □

The final step is to show strong duality for linear programming, that is, that the objective values of (3.4) and (3.16) are equal when both problems are solved to optimality. For this we use the following lemma from [5, pp. 232–234].

Lemma 1. *Let $P := \{x \in \mathbb{R}^n \mid Ax = b,\ x \geq 0\}$ be a convex polyhedron and $V := \{v^1, \ldots, v^k\}$ be the extreme points of P, see definition 1.*

Now consider the linear program

$$\begin{aligned} \min_{x} \quad & z = c^\top x, \\ \textit{subject to} \quad & x \in P. \end{aligned} \tag{3.17}$$

Then,

1. (3.17) *has a finite optimal solution if and only if P is nonempty and z is bounded from below on P.*
2. *If* (3.17) *has a finite optimal solution, then there exists an optimal solution among the extreme points to P.*

Proof. See [5, pp. 233–234]. □

This result ensures the existence of optimal solutions for LP problems of interest to this work, that is, for finite objective values.

Strong duality for linear programming now follows:

Theorem 4. *If there exists feasible solutions x and y to* (3.4) *and* (3.16)*, respectively, then there exists optimal solutions $\hat{x}$ and $\hat{y}$ to the respective problems and $c^\top \hat{x} = b^\top \hat{y}$ holds.*

Proof. Since y is feasible in (3.16) it follows from Theorem 3 that the objective z in (3.4) is bounded from below. By Lemma 1 we then have that there exists an optimal solution to (3.4) among the extreme points of its feasible region. Since the extreme points are equivalent to a BFS, the optimal solution is then $\hat{x} = (\hat{x}_B^\top, \hat{x}_N^\top)^\top$, where B and N determine a BFS given by the solution $\hat{x}$.

Let (3.5) define a partitioning of the primal problem (3.4) with the associated optimal basis B. Now set

$$\hat{y}^\top = c_B^\top B^{-1}. \tag{3.18}$$

Since $\hat{x}$ is optimal in (3.4), the reduced costs corresponding to the (optimal) basis B, are non-negative. That is,

$$c_N^\top - c_B^\top B^{-1} N = c_N^\top - \hat{y}^\top N \geq 0. \tag{3.19}$$

Also, $c_B^\top - \hat{y}^\top B = c_B^\top - c_B^\top B^{-1} B = 0$ holds, giving

$$c^\top - \hat{y}^\top A \geq 0 \iff A^\top \hat{y} \leq c.$$

Thus, $\hat{y}$ as defined in (3.18) is feasible in (3.16). Finally, by (3.7), we conclude that

$$b^\top \hat{y} = b^\top (B^{-1})^\top c_B = c_B^\top B^{-1} b = c_B^\top \hat{x}_B = c_B^\top \hat{x}_B + c_N^\top \hat{x}_N = c^\top \hat{x},$$

which by Theorem 3 yields that $\hat{y}$ is optimal in (3.16). □

This result will be used to compute bounds on the optimal objective value in the MBLP model to be formulated in later Section 4; such bounds are important aspects of many solution strategies, e.g., column generation (see Section 3.8).

3.2 Logical Constraints

In an optimization problem it is sometimes necessary to make a decision regarding quantities that need to be whole units. For example, if workers are to be assigned jobs at a station in a manufacturing plant, then it does not make sense to assign the two halves of one worker to two different stations. For LP problems the feasible region is made up of constraints in the form of inequalities (and equalities) expressed as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots a_{1n}x_n &\leq b_1, \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots a_{mn}x_n &\leq b_m. \end{aligned} \tag{3.20}$$

In a feasible solution, the decision variable $x \in \mathbb{R}^n$ can take any real number, as long as $x \geq 0$ is fulfilled together with the m linear inequalities in (3.20). But to make "yes" or "no" decisions there is a need for introducing additional constraints that utilize binary variables $w \in \{0,1\}$, such that

$$w = \begin{cases} 1, & \text{if a positive decision is to be made,} \\ 0, & \text{if a negative decision is to be made.} \end{cases}$$

To formulate logical constraints using this type of variables, boolean algebra is a powerful tool. In Table 3.1, some logical connectives and translations into affine constraints are presented. Using these logical connectives makes it is easier to formulate combinatorical

Logical expression	**Interpretation**	**Constraint**
$E_1 \wedge E_2$	both E_1 and E_2	$\delta_1 + \delta_2 = 2$
$E_1 \vee E_2$	E_1 or E_2 or both	$\delta_1 + \delta_2 \geq 1$
$E_1 \bar{\vee} E_2$	exactly one of E_1 and E_2	$\delta_1 + \delta_2 = 1$
$(E_1 \bar{\vee} E_2) \bar{\vee} \neg(E_1 \vee E_2)$	one of E_1, E_2 or none	$\delta_1 + \delta_2 \leq 1$
$E_1 \Rightarrow E_2$	if E_1 then E_2	$\delta_1 - \delta_2 \leq 0$
$E_1 \Leftrightarrow E_2$	E_1 if and only if E_2	$\delta_1 - \delta_2 = 0$

Table 3.1: Logical forms with constraints. Here, E_1 and E_2 are events that can be realized by binary decision variables δ_1 and δ_2.

constraints of type "if E_1 then E_2" for events E_1 and E_2. A typical case could be if a vehicle routing schedule is to be made and one wishes to prohibit multiple vehicles from taking the same route. For more details and examples of binary variables and logical forms, see [6, Section 6.3].

3.3 Network Flow Modelling

Typical applications of integer or binary variables in optimization modelling, are network models. Some problems can be formulated in terms of a graph with nodes and arcs. The *assignment problem* is an example of this. The nodes are then associated with either a task or a resource to perform the task (worker/machine/processor etc). The arcs between tasks and resources are associated with the cost that it takes for the specific resource to perform the task. The problem is to find an assignment of resources to tasks, such that the cost is minimized. The restriction on the assignment is that every task is to be carried out, but only by one of the resources and that each resource is to perform exactly one task. Figure 3.2 shows the network graph of an assignment problem, with the costs left out. If we consider a more general assignment problem where all tasks $i = 1, 2, \ldots, N$,

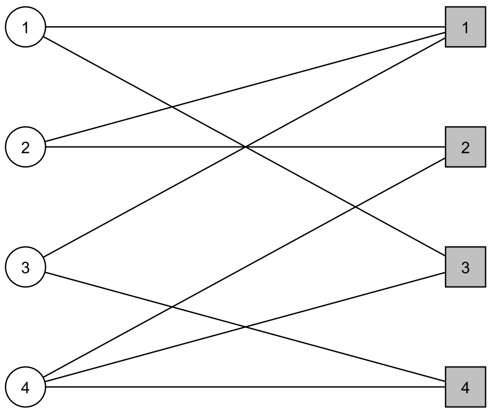


Figure 3.2: Assignment problem graph.

can be carried out by any resource $j = 1, 2, \ldots, N$, the following notations can be used to formulate a model of the assignment problem. Let the costs for a resource i to perform task j be $c_{i,j}$ and introduce $y_{i,j}$ as

$$y_{i,j} = \begin{cases} 1, & \text{if resource } i \text{ is assigned to task } j, \\ 0, & \text{otherwise.} \end{cases}$$

A model of the assignment problem can then be stated as

$$\begin{aligned}
\min_{y} \quad & \sum_{i=1}^{N}\sum_{j=1}^{N} c_{i,j}y_{i,j}, && \\
\text{subject to} \quad & \sum_{i=1}^{N} y_{i,j} = 1, && j = 1,2,\dots,N, \\
& \sum_{j=1}^{N} y_{i,j} = 1, && i = 1,2,\dots,N, \\
& y_{i,j} \in \{0,1\}, && i = 1,2,\dots,N, \\
& && j = 1,2,\dots,N.
\end{aligned} \tag{3.21}$$

The first constraint ensures that each task is carried out by exactly one resource, and the second constraint requires every resource to carry out precisely one task.

Another example of a problem with a more obvious network structure is the *minimum cost network flow problem.* This problem is typical in, for example, delivery routing, where a commodity is to be delivered from a source to a depot, while minimizing the cost of delivery. The nodes represents stops on the route and each arc has an associated cost and a capacity. The problem is to find a flow of the commodity from a source node to a sink node, such that capacities and flow conservation restrictions are met, while minimizing the total cost of the flow. Figure 3.3 shows an example of a network flow graph with costs and capacities on the arcs, denoted by (cost, capacity).

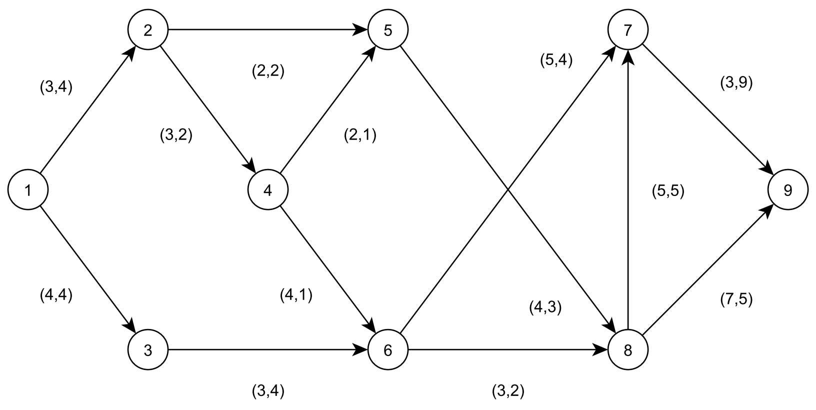


Figure 3.3: A network flow graph with costs and capacities on the arcs.

For a general problem with N nodes, denote the amount of flow from node i to j, by $x_{i,j}$. The costs are, similar to the assignment problem, $c_{i,j}$ and the capacities $k_{i,j}$. An initial flow from the source node 1 is denoted f_0, which is to be received by the sink node N. The model can now be formulated as

$$\begin{aligned}
\min_{x} \quad & \sum_{i=1}^{N}\sum_{j=1}^{N} c_{i,j}x_{i,j}, \\
\text{subject to} \quad & \sum_{j=1}^{N} x_{1,j} = f_0, \\
& -\sum_{i=1}^{N} x_{i,N} = -f_0, \\
& -\sum_{j=1}^{N} x_{i,j} + \sum_{j=1}^{N} x_{j,i} = 0, \qquad i = 2,\dots,N-1, \\
& 0 \le x_{i,j} \le k_{i,j}, \qquad i,j = 1,2,\dots,N, \\
& x_{i,j} \in \mathbb{Z}, \qquad i,j = 1,2,\dots,N.
\end{aligned} \tag{3.22}$$

The first and second constraint require that the entire flow of the commodity originates from the source and is received by the sink. The third constraint is a flow-conservation constraint. It ensures that for every intermediate node i that sends a flow of the commodity, it is required that the corresponding node j receives the same amount. The final constraint is to prohibit the flow from exceeding the capacity of the links between nodes.

A more general version of the above flow problem, is when there are multiple commodities that should be sent through the network. In that case a third index is added to the flow variables $x_{i,j}$ in (3.22), specifying the type of commodity, and denoted $x_{i,j,q}$, where q denotes the commodity. The above flow constraints should then hold for all commodities. The capacity constraint should then limit the sum of all commodities, i.e., it should read $\sum_{q\in Q} x_{i,j,q} \le k_{i,j}$. In some cases the integrality constraints are dropped and the commodity flows are allowed to take on fractional values.

The problem attacked in this work is more or less a combination of the above mentioned problems. Signal processing functions are to be assigned to a DSP that should carry out a computational task, and the connections between DSPs can be interpreted as possessing flow-conservation properties, where there is a capacity limitation on the number of connections that can be used per DSP. Hence, many of the constraints formulated in Section 4 will resemble the constraints in (3.21)–(3.22).

3.4 Mixed-Binary Linear Programming

The differences between a general MBLP model and an LP model can, in the aspects relevant to this work, be summarized by those between ILP models (integer linear programming models) and LP models. More precisely, the geometric properties of the feasible region of an ILP model and an MBLP model are similar, as compared to an LP model.

Adding to that, the method of solving these models, the branch-and-bound method, is used for both MBLP- and ILP models. To facilitate the presentation of notations and figures, the contents of this section will therefore focus on ILP models.

A general ILP model can be formulated as

$$\min_{x} \quad z = c^\top x, \tag{3.23a}$$
$$\text{subject to} \quad Ax \leq b, \tag{3.23b}$$
$$x \geq 0, \tag{3.23c}$$
$$x \in \mathbb{Z}_+^n \tag{3.23d}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$. Using the same example region as in Figure 3.1, with the integrality constraints added, the new feasible region is shown in Figure 3.4. Note that the uppermost extreme point in Figure 3.1 is not feasible in the corresponding ILP problem. This is generally the case when the integrality constraints (3.23d) are added. When the decision variables are required to be integers, the convexity of the feasible region is lost, and extreme points, as defined in Definition 1, do not exist. This is one of the reasons why the simplex method cannot be used to solve general ILP problems.

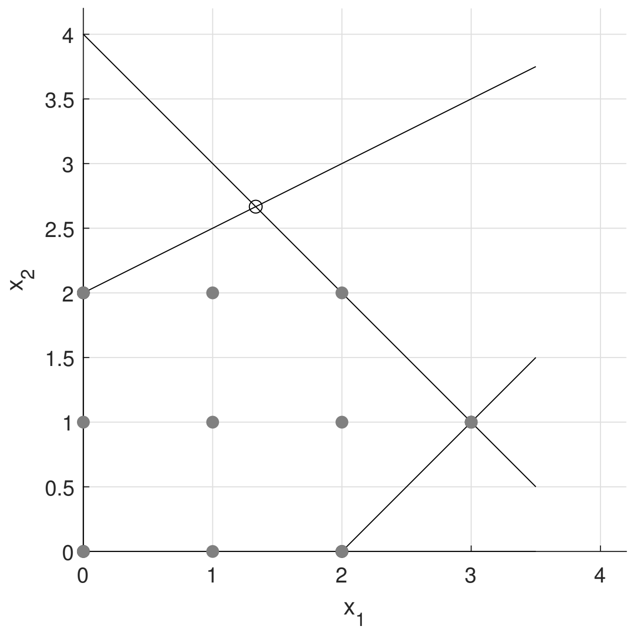


Figure 3.4: The feasible region of an ILP. Filled grey circles represents the feasible points.

The models presented in Section 3.3, are both cases of ILP models, and the model created for the mapping of processing functions in this work, will also be of this kind. The feasible region will therefore be similar to the one presented in Figure 3.4, but in higher dimensions.

Removing the integrality constraints (3.23d) would yield its *LP-relaxation.* The LP model can be solved efficiently using the simplex method and an optimal value with corresponding decision variables can be found. An intuitive method of finding a feasible solution to the ILP model would then be to round the decision variables down to integer numbers. However, this solution may very well be far from optimal in the ILP model. In [6, pp. 211–210], Sierksma outlines why this procedure can fall short and motivates why other methods are needed.

3.5 Branch-and-Bound Method

The LP-relaxation of an ILP model is a relaxation in terms of (3.10) and (3.11). By Theorem 1, solving the LP-relaxation of an ILP model produces a lower bound on the optimal objective value of the ILP. This fact is the main building block of the branch-and-bound (BnB) method. Roughly, the method searches the feasible region of the ILP, by solving instances of LP-relaxations and comparing the change in the objective value at each iteration. At every iteration the algorithm solves an LP-relaxation of the current parent model and divides it into submodels with smaller feasible regions by adding constraints on the decision variables taking non-integer values. The dividing of the feasible region is called *branching*, and the comparing with the objective value of the LP-relaxation is called *bounding.*

The algorithm starts with solving the LP-relaxation of the original ILP formulation, giving a lower bound on the optimal objective value of the ILP model. If all the decision variables take on integer-values, the current solution is optimal (again by the Relaxation Theorem). If this is not the case, every non-integer valued variable starts a *branching (sub-)tree.* The first nodes in each branch consist of two submodels for each variable that initialized the branch.

For example, we start by solving an initial parent model, the LP-relaxation of some ILP formulation. Assume that in this solution, one of the variables $x = 10.2$. Then two submodels of the original formulation are created, where the constraints

$$x \geq 11,$$

and

$$x \leq 10,$$

are added to the respective submodel. A branching on the variable x has been made. In the case of binary variables, the branching will instead be to restrict non-binary valued variables to 1 or 0.

The next step is to choose one of these submodels as the new parent model, solve the LP-relaxation of this submodel and calculate the objective value. If a submodel is infeasible or if the solution happens to be integer, no further branching will be made on that model. If the solution to a submodel is feasible with non-integer valued variables, a new branching tree is rooted for one of these variables, where constraints are added on the variables as above.

By repeating this procedure, the feasible region shrinks further down the tree. This means that in every branch created, the objective values of previously calculated solutions, can be used to conclude that some branching will be unnecessary, and the branch can be "cut". E.g, a non-integer solution with larger objective value than an integer solution in the same branch, does not need to branched upon further.

An example of a branch-and-bound tree is given in Figure 3.5. The submodels are denoted by S and the branching is made on the two variables x and y in this case. The remaining variables are not shown. At every node S_i one can choose to branch on any non-integer valued variable, and therefore the trees can grow very large, very quickly.

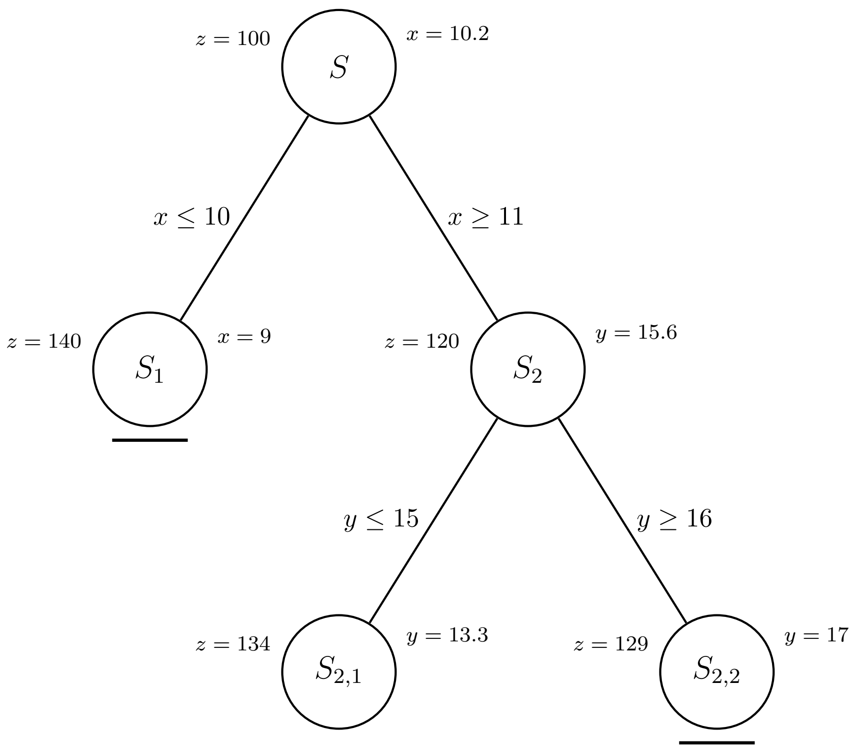


Figure 3.5: A branch-and-bound tree. Bars below the nodes indicate integer solutions.

In the node S_1 every variable is integer in the LP-relaxation of the submodel, so no further branching is done. The branching in $S_{2,1}$ is also stopped, since the integer solution in $S_{2,2}$ has a better (lower) objective value, and by the Relaxation Theorem, the objective in $S_{2,1}$ will not improve as more constraints are added to this submodel. A general description of the algorithm, complete with selection rules for branching and examples of solving MILP models by the algorithm, can be found in [6, Section 6.2.2].

The branch-and-bound algorithm is highly parallelizable, since all submodels can be solved independently. Modern multicore-processors have therefore lead to a significant decrease in computational running time for ILP-programs. Also, the simplex method, which is often used to solve the submodels, is an efficient algorithm. However, the computational complexity of the branch-and-bound algorithm indicates that for large-scale MILP problems an alternative method is needed.

3.6 Computational Complexity

When choosing the solution method of an optimization problem, it is important to consider the computational complexity of the method. In this section a brief summary of some of the key concepts and theory introduced by Schrijver [7], will be presented. This will give incentive to the column generation approach made in this work.

The most basic concepts in computational complexity theory are *problems* and *size*, *algorithms*, and *running time*. There are different ways of defining a problem, but an intuitive form is that of a question. A BLP problem can be formulated as:

$$\begin{array}{l}\text{Given a rational matrix } A \in \mathbb{R}^{m \times n} \text{ and a rational vector } b \in \mathbb{R}^m, \\ \text{does } Ax = b \text{ have a solution } x \in \{0,1\}^n?\end{array} \tag{3.24}$$

Formally, this question is called a *decision problem.* The parameters A and b of the problem are the input to an *algorithm* which will, if possible, produce an output x that answers the question. An algorithm itself is defined through a computing model, in most cases a variant of a *Turing Machine.* A Turing Machine is informally an abstract machine that executes operations depending on the input that it currently reads. Theoretically, it resembles a computer. Simply put, an algorithm is a list of instructions to solve a problem. How the number of operations—that the machine needs to execute to complete the instructions—scales with the problem size is measured by the *running time* of the algorithm. The *problem size* is thus defined through the number of instructions needed to solve a problem and these instructions are made up of elementary arithmetic operations such as adding, subtracting, multiplying, dividing, and comparing numbers. The number of such elementary operations n, will be referred to as the problem size. The running time is then given by a function $f(n)$.

To compare and classify running times, big-O notation is most commonly used.

Definition 3. (big-O) Let $f : S \mapsto \mathbb{R}$ and $g : S \mapsto \mathbb{R}$ be functions, with $S \subseteq \mathbb{R}_+$ being unbounded and $g(x) \geq 0$ for $x \in S$, then

$$\lim_{x\to\infty} f(x) = O(g(x)),$$

if and only if there exists $M \geq 0$ and $x_0 \in S$ such that

$$|f(x)| \leq Mg(x), \quad \text{for } x \geq x_0.$$

One says that "$f(x)$ is $O(g(x))$" if it is upper bounded by the function $g(x)$. In terms of running time $f(n)$ and problem size $n \in \mathbb{Z}_+$, one says an algorithm runs in $O(g(n))$-time.

3.7 NP-Completeness

Problems, such as (3.24), are classified by the running time of the the algorithm used to solve them. Running times can be compared in a worst-case sense or in an average sense. Worst-case would refer to the largest running time possible for a given problem size n. It is in general accepted to estimate times by worst-case [6, pp. 218–219], since this is in general a simpler procedure.

The two classes which will be the focus here, are P and NP. Problems that belong to P are solved in polynomial running time. That is, the running time $f(n)$ is upper bounded by a polynomial $g(n)$. These problems are considered being "easy" and efficient algorithms can solve them "quickly". A classic example of a problem in P is the dot product of two vectors $a, b \in \mathbb{R}^n$. To solve this problem, n multiplications and $n-1$ additions are needed. Thus, $f(n) = n + n - 1 = 2n - 1$, which is $O(n)$.

For the problems in NP, no known algorithm exists by which they can be solved in polynomial time. An example of a problem in NP is the 0-1 knapsack problem (with unit cost coefficients):

$$\begin{aligned} \max_{x} \qquad & z = \sum_{i=1}^{n} x_i, \\ \text{subject to} \qquad & \sum_{i=1}^{n} a_i x_i \leq K, \\ & \qquad x \in \{0,1\} \end{aligned} \tag{3.25}$$

Solving this problem using the branch-and-bound algorithm can result in a running time which is exponential in n. Sierksma [6, pp. 326–328], shows that for an example similar to (3.25) (with $K \geq 3$ and odd, and $a_i = 2$ for all i) yields that the number of submodels in the branch-and-bound tree becomes $2^{\frac{n+1}{2}}$, in a worst-case scenario. Bare in mind that this is just the number of submodels, and to solve them many further instructions are needed. The problems in NP are therefore considered hard to solve.

In NP there are subgroups of problems considered the hardest in the class. These are called NP-complete problems. A problem Q is called NP-complete if it belongs to NP and there exists an algorithm for which any other problem in NP can be *reduced* (see the definition of *Karp-reduction* [7, pp. 20–21]) to Q in polynomial time. This means that any algorithm that solves an NP-complete problem can be used to solve any problem in NP. This also implies that if there exists an algorithm which solves an NP-complete problem in polynomial time, then all problems in NP can be solved in polynomial time and $P = NP$. However, such an algorithm is yet to be found.

Schrijver [7, Section 18] gives extensive proofs that the problem (3.24), along with a number of combinatorial and ILP problems, are all NP-complete problems. These include the integer (or 0-1) Knapsack problem and the Travelling Salesperson problem. For large-scale mapping of processing functions, this becomes a practical issue. An attempt at resolving this is through column generation.

3.8 Problem Decomposition and Column Generation

Column generation is a method applied to optimization problems when the number of feasible solutions is too large for a search-based method, such as the branch-and-bound algorithm. The idea behind it is to consider only a smaller subset of the solutions, i.e, columns, at a time. The method is designed for LP problems, when it is too expensive (in terms of computational running time) to calculate the reduced costs (see Section 3.1.1) for all the non-basic variables. When using the method for ILP problems (or MBLP problems in this case) it is therefore necessary to first decompose the problem into an ILP part and an LP part. This is called Dantzig-Wolfe decomposition and will be the first step to applying the column generation method. Most of this section will follow that of [4], where more details and variants of implementations can be found.

3.8.1 Dantzig-Wolfe Decomposition

Consider a BLP with two sets of constraints

$$\begin{aligned}
\min_{x} \qquad & z = c^\top x, && (3.26a)\\
\text{subject to} \qquad & Ax \leq b, && (3.26b)\\
& Dx \leq d, && (3.26c)\\
& x \in \{0,1\}^n, && (3.26d)
\end{aligned}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, $D \in \mathbb{R}^{l\times n}$ and $d \in \mathbb{R}^l$. This will be referred to as the *Original Formulation.* Let

$$X := \{x \in \{0,1\}^n \mid Dx \leq d\} \neq \emptyset. \tag{3.27}$$

Moving on, we will use results from [7, Section 7.1–7.2], regarding the decomposition of polyhedral sets, such as X. The Minkowski-Weyl theorems state that a polyhedron H can be expressed as a convex combination of the extreme points and extreme rays of its convex hull, conv(H).

Definition 4 (Convex Hull)**.** The convex hull of a set $H \subseteq \mathbb{R}^n$, $\operatorname{conv}(H)$, is the smallest convex set containing the set H.

No further attention will be given to extreme rays of polyhedra, since for x binary, X can be expressed through the extreme points x_p of $\operatorname{conv}(X)$ alone. Thus,

$$\begin{aligned} \sum_{p \in \mathcal{P}} \lambda_p x_p &= x, \\ \sum_{p \in \mathcal{P}} \lambda_p &= 1, \\ \lambda_p &\geq 0, \text{ for } p \in \mathcal{P}, \end{aligned} \tag{3.28}$$

where $\mathcal{P}$ is the set indexing the extreme points. Using this representation of x in (3.26) yields the equivalent formulation

$$\min_{x} \quad z_{\text{IMP}} := \sum_{p \in \mathcal{P}} \lambda_p (c^\top x_p), \tag{3.29a}$$

$$\text{subject to} \quad \sum_{p \in \mathcal{P}} \lambda_p (A x_p) \leq b, \tag{3.29b}$$

$$\sum_{p \in \mathcal{P}} \lambda_p = 1, \tag{3.29c}$$

$$\lambda_p \geq 0, \qquad \text{for } p \in \mathcal{P}, \tag{3.29d}$$

$$x = \sum_{p \in \mathcal{P}} \lambda_p x_p, \tag{3.29e}$$

$$x \in \{0,1\}^n. \tag{3.29f}$$

The formulation (3.29) is called the *Integer Master Problem* (IMP). The constraints (3.29c)–(3.29d) are called the *convexity constraints*. Relaxing the integrality constraint on x in (3.29f), gives the LP model suitable for column generation. The columns $\left(c^\top x_p, (Ax_p)^\top\right)$, are then defined through the extreme points x_p to the set X as defined in (3.27). When the integrality constraints are no longer present, the constraint (3.29e) is neither needed. The LP-relaxed IMP is then given by

$$\min_{\lambda} \quad z_{\text{MP}} := \sum_{p \in \mathcal{P}} \lambda_p (c^\top x_p), \tag{3.30a}$$

$$\text{subject to} \quad \sum_{p \in \mathcal{P}} \lambda_p (A x_p) \leq b, \tag{3.30b}$$

$$\sum_{p \in \mathcal{P}} \lambda_p = 1, \tag{3.30c}$$

$$\lambda_p \geq 0, \quad \text{for } p \in \mathcal{P}. \tag{3.30d}$$

This formulation will be referred to as the *Master Problem* (MP). It is the LP part of the original problem (3.26), and depends only on the variables λ_p, of which there is one for every column.

3.8.2 Column Generation

For X large enough, it is impractical to enumerate all extreme points x_p, in order to solve (3.30). Instead, one can apply column generation, and only consider a limited number of variables at a time. The restricted subset of columns which are considered is denoted $\mathcal{P}' \subseteq \mathcal{P}$ and the task is now to optimize the *Restricted Master Problem* (RMP)

$$\begin{aligned}
\min_{\lambda} \quad & z_{\text{RMP}} := \sum_{p \in \mathcal{P}'} \lambda_p (c^\top x_p), && (3.31\text{a}) \\
\text{subject to} \quad & \sum_{p \in \mathcal{P}'} \lambda_p (A x_p) \leq b, && (3.31\text{b}) \\
& \sum_{p \in \mathcal{P}'} \lambda_p = 1, && (3.31\text{c}) \\
& \lambda_p \geq 0, \quad \text{for } p \in \mathcal{P}'. && (3.31\text{d})
\end{aligned}$$

In column generation, like in the simplex method, promising variables are searched for to enter an optimal basis; see Section 3.1.1. But as opposed to using the simplex method, where the reduced costs are calculated for all non-basic variables, one searches for a single variable λ_p, $p \in \mathcal{P} \setminus \mathcal{P}'$, which still has a negative reduced cost. If there exists such a variable, it is added to the RMP, together with its corresponding column $\left(c^\top x_p, (Ax_p)^\top\right)$. The updated RMP is then re-optimized, using, e.g., the simplex method.

As shown in the proof of Theorem 4, the calculation of reduced costs for non-basic variables can be done through the dual variables. The reduced cost for a variable λ_p, can therefore be expressed as

$$\tilde{c_p} := (c^\top - \pi^\top A) x_p - \pi_0,$$

where $\pi \in \mathbb{R}^m$ and $\pi_0 \in \mathbb{R}$ are the dual variable values calculated from the solution of the current RMP, corresponding to constraints (3.31b) and (3.31c), respectively. The search for a variable λ_p with the most negative reduced cost then takes the form of solving a *subproblem*

$$\begin{aligned}
\min_{p} \quad & (c^\top - \pi^\top A) x_p - \pi_0, \\
\text{subject to} \quad & p \in \mathcal{P},
\end{aligned} \qquad (3.32)$$

or equivalently

$$\begin{aligned}
\min_{x} \quad & (c^\top - \pi^\top A) x - \pi_0, \\
\text{subject to} \quad & x \in X,
\end{aligned} \qquad (3.33)$$

which can be solved by the branch-and-bound method. If $\tilde{c}_p \geq 0$ in the current iteration of the subproblem, then the RMP and the MP are optimal and equivalent, and no further columns need to be added. The integer constraints can then be reinstated on the convexity variables λ_p in the RMP (forming the integer RMP, or IRMP), and a feasible solution among the generated columns can then be searched for, again by branch-and-bound. In general, however, there might not exist a feasible solution to the original problem among the columns.

To summarize, an iteration of the column generation algorithm involves optimizing the current RMP (using the simplex method) to compute dual variable values which can be used to search for a new column via the subproblem. When optimizing the subproblem through branch-and-bound, the variable with the most negative reduced cost is found and added to the RMP, together with its associated column. The RMP is then re-optimized, and the process starts over. When the reduced cost $\tilde{c}_p \geq 0$, the MP is optimal and a feasible solution to the original formulation can be searched for among the columns.

The process can be initiated with a feasible solution to the original problem, as the first column. An artificial variable with a large cost can then be used to find a first feasible solution to the RMP, and corresponding dual variable values can be calculated. With this initiation, one is guaranteed a feasible solution (however, bad it might be) when the column generation is complete, i.e., the first column. An improvement to the original feasible solution might then exist in the the columns generated.

3.8.3 Block-Diagonal Structure

For certain problems, a block-diagonal structure can be present in the matrix D in (3.27). That is, the constraints (3.26c) can be expressed as

$$\begin{pmatrix} D^1 & & & & \\ & D^2 & & 0 & \\ & & \ddots & & \\ & 0 & & D^{K-1} & \\ & & & & D^K \end{pmatrix} \begin{pmatrix} x^1 \\ x^2 \\ \vdots \\ x^{K-1} \\ x^K \end{pmatrix} = \begin{pmatrix} d^1 \\ d^2 \\ \vdots \\ d^{K-1} \\ d^K \end{pmatrix}.$$

The feasible set of the subproblem can then be divided into K subsets, belonging to different subspaces, as

$$X^k := \{x^k \in \{0,1\}^{|d^k|} \mid D^k x^k \leq d^k\},$$

with $k \in \mathcal{K} = \{1, 2, \dots, K\}$, and one instead solves K subproblems

$$\begin{aligned} \min_{x^k} \quad & (c_k^\top - \pi^\top A_k)x^k - \pi_0^k, \\ \text{subject to} \quad & x^k \in X^k. \end{aligned}$$

Depending on the distribution of constraints among the K subproblems, they might differ in complexity and required computational effort.

In the RMP, variables λ_p^k, with $p \in \mathcal{P}'_k$, $k \in \mathcal{K}$, are added when the reduced cost $\tilde{c}_p^k < 0$ for any of the subproblems k, i.e.,

$$\begin{aligned} \min_{\lambda} \quad & z_{\text{RMP}} = \sum_{k\in\mathcal{K}}\sum_{p\in\mathcal{P}'_k} \lambda_p^k (c^\top x_p^k), \\ \text{subject to} \quad & \sum_{k\in\mathcal{K}}\sum_{p\in\mathcal{P}'_k} \lambda_p^k (A x_p^k) \leq b, \\ & \sum_{p\in\mathcal{P}'_k} \lambda_p^k = 1, \quad \text{for } k\in\mathcal{K}, \\ & \lambda_p^k \geq 0, \quad \text{for } p \in \mathcal{P}'_k, \quad k\in\mathcal{K}. \end{aligned} \tag{3.34}$$

This gives rise to K convexity constraints, and associated dual variables π_0^k, in the subproblems. The column generation ends when $\tilde{c}_p^k \geq 0$ for all $k \in \mathcal{K}$. An advantage of this structure is that the subproblems can be solved independently, giving the option to parallelize the implementation of the column generation algorithm.

3.8.4 Lagrangian Bounds

A convenient fact about the column generation method is that upper and lower bounds on the optimal value can be calculated for the MP (3.30), every time the RMP is optimized. Hence, the quality of the solution and the progress of the algorithm, can be evaluated in each iteration.

The number of extreme points to the set X considered in the RMP is restricted, and so the feasible region is smaller compared to the MP. By the Relaxation Theorem, this gives an upper bound on the optimal value $\hat{z}_{\text{MP}}$ of the MP, using the optimal value $\hat{z}_{\text{RMP}}$ of the current instance of the RMP, according to

$$\hat{z}_{\text{MP}} \leq \hat{z}_{\text{RMP}}.$$

This upper bound will decrease monotonically as the column generation progresses, and the above inequality will be fulfilled with equality when all necessary columns have been generated. This is because at every iteration only variables with negative reduced costs are added to the RMP, and consequently the optimal objective value decreases. Since the dual variables from the RMP are used to solve the subproblem, strong duality for linear programs can be used to compute this bound. Consider the dual problem of the RMP,

$$\max_{(\pi,\pi_0)} \quad b^\top \pi + \pi_0, \tag{3.35a}$$

$$\text{subject to} \quad (A x_p)^\top \pi + \pi_0 \leq c^\top x_p, \quad p \in \mathcal{P}', \tag{3.35b}$$

$$\pi \leq 0, \tag{3.35c}$$

with optimal solution $(\hat{\pi}, \hat{\pi}_0)$. This gives

$$\hat{z}_{\text{MP}} \leq b^\top \hat{\pi} + \hat{\pi}_0 = \hat{z}_{\text{RMP}}.$$

A lower bound on the MP can be found by considering the optimal $\hat{\lambda}_p$ in the MP and the above calculated bound, according to

$$\begin{aligned}
0 \geq \hat{z}_{\text{MP}} - b^\top \hat{\pi} - \hat{\pi}_0 &= \sum_{p \in \mathcal{P}} \hat{\lambda}_p (c^\top x_p) - b^\top \hat{\pi} - \hat{\pi}_0 && \text{(3.36a)}\\
&\geq \sum_{p \in \mathcal{P}} \hat{\lambda}_p (c^\top x_p) - \sum_{p \in \mathcal{P}} \hat{\lambda}_p (A x_p)^\top \hat{\pi} - \sum_{p \in \mathcal{P}} \hat{\lambda}_p \hat{\pi}_0 && \text{(3.36b)}\\
&= \sum_{p \in \mathcal{P}} \hat{\lambda}_p \Big[c^\top x_p - (A x_p)^\top \hat{\pi} - \hat{\pi}_0 \Big] && \text{(3.36c)}\\
&\geq \min_{p \in \mathcal{P}} \Big[c^\top x_p - (A x_p)^\top \hat{\pi} - \hat{\pi}_0 \Big] && \text{(3.36d)}\\
&= \min_{x \in X} (c^\top - \pi^\top A) x - \pi_0, && \text{(3.36e)}
\end{aligned}$$

where $\sum_{p \in \mathcal{P}} \hat{\lambda}_p = 1$ (3.30c) was used as the coefficient for $\hat{\pi}_0$ in (3.36b), and the constraint $\sum_{p \in \mathcal{P}} \hat{\lambda}_p (A x_p)^\top \leq b^\top$ (3.30b) was used together with $\hat{\pi} \leq 0$ (3.35c) to get the inequality in (3.36b). The inequality in (3.36d) is also due to the constraint $\sum_{p \in \mathcal{P}} \hat{\lambda}_p = 1$ in (3.30c). The last equality can be recognized as the minimal reduced cost from the subproblem (3.33).

Thus, $\hat{z}_{\text{MP}}$ is bounded from above and below by the inequalities

$$\hat{z}_{\text{RMP}} + \min_{x \in X} (c^\top - \pi^\top A) x - \pi_0 \leq \hat{z}_{\text{MP}} \leq \hat{z}_{\text{RMP}}. \tag{3.37}$$

In the case of a block-diagonal structure, the lower bound instead becomes

$$\hat{z}_{\text{RMP}} + \sum_{k \in \mathcal{K}} \min_{x^k \in X^k} \left\{ (c^\top - \pi^\top A) x^k - \pi_0^k \right\} \leq \hat{z}_{\text{MP}}. \tag{3.38}$$

Since the integer master problem (3.29) is restricted to binary variables, the inequality $\hat{z}_{\text{MP}} \leq \hat{z}_{\text{IMP}}$ holds and the lower bound in (3.37) for the MP, is also a lower bound for the IMP. But the upper bound only holds when the constraints (3.29f) and (3.29e) holds in the RMP, as well. These bounds can now be calculated in each iteration of the column generation, and a termination criterion can be set using the difference between the upper and lower bound, i.e., the optimal objective value of the subproblem. Thus, the progress of the algorithm can be evaluated at each iteration.

4

Formulation of Models

In Chapter 2 the specifics of the problem at hand are given. This includes the function sequence framework to describe the processing graph, and some details around the many-core grid to which the processing functions should be mapped. In this chapter, some of the modelling techniques presented in Section 3.2 will be used to form a MBLP model of the problem, which can then be solved by the branch-and-bound algorithm. The MBLP model will then serve as the original formulation (see (3.26)) for two different Dantzig-Wolfe decompositions from which approximately optimal solutions can be found through the column generation method.

4.1 MBLP Model

First, the variables of the problem will be defined. These will then be used to formulate the objective subject to minimization, and the problem restrictions in terms of logical constraints.

Define the following sets:

$\mathcal{I}$: an ordered set of rows on the grid; $|\mathcal{I}| = I$

$\mathcal{J}$: an ordered set of columns on the grid; $|\mathcal{J}| = J$

$\mathcal{M}$: the set of function sequences; $|\mathcal{M}| = M$

$\mathcal{N}_m$: the ordered set of functions in sequence $m \in \mathcal{M}$; $|\mathcal{N}_m| = N_m$

$\mathcal{T}$: the set of connection nodes in the processing graph

$\mathcal{M}_t^N$: the set of sequences ending in connection node $t \in \mathcal{T}$; $\mathcal{M}_t^N \subseteq \mathcal{M}$

$\mathcal{M}_t^1$: the set of sequences starting in connection node $t \in \mathcal{T}$; $\mathcal{M}_t^1 \subseteq \mathcal{M}$

$\mathcal{M}_{\text{input}}$: the set of function sequences containing input functions; $\mathcal{M}_{\text{input}} \subset \mathcal{M}$

$\mathcal{M}_{\text{output}}$: the set of function sequences containing output functions; $\mathcal{M}_{\text{output}} \subset \mathcal{M}$

Initially, a BLP model will be formulated, where two sets of binary variables are being defined to describe the mapping of, and the coupling between, functions. Thus, define the assignment variables for functions as

$$y_{m,n,(i,j)} := \begin{cases} 1, & \text{if the function } f_{m,n} \text{ is} \\ & \text{assigned to position } (i,j), \\ 0, & \text{otherwise,} \end{cases} \qquad \begin{array}{c} (i,j) \in \mathcal{I} \times \mathcal{J}, \\ n \in \mathcal{N}_m,\ m \in \mathcal{M}, \end{array}$$

and the coupling variables

$$x_{m,n,(i,j),(q,r)} := \begin{cases} 1, & \text{if the coupling } (f_{m,n}, f_{m,n+1}) \text{, passes link } ((i,j),(q,r)), \\ 0, & \text{if not,} \end{cases} \tag{4.1}$$

where $(q,r) \in \{(i-1,j),(i+1,j),(i,j-1),(i,j+1)\} \cap \mathcal{I} \times \mathcal{J} := \mathcal{V}(i,j)$. The set $\mathcal{V}(i,j)$ restricts the coupling variables to use only horizontal and vertical links between neighbouring DSPs on the grid. Defining $\mathcal{V}(i,j)$ through the intersection as above results in that the elements corresponding to the missing neighbour positions on the edges of the grid are not present in the set. The coupling variables describe the network connection path between DSPs used by sequential functions. Each variable $x_{m,n,(i,j),(q,r)}$, defined in (4.1), is associated with the locations (i,j) and (q,r), since these positions also constitute switches which can be used to route data.

The objective to minimize is the number of couplings between the functions mapped to the grid, resulting in the objective function

$$z := \sum_{m \in \mathcal{M}} \sum_{n \in \mathcal{N}_m} \sum_{(i,j) \in \mathcal{I} \times \mathcal{J}} \sum_{(q,r) \in \mathcal{V}(i,j)} x_{m,n,(i,j),(q,r)}. \tag{4.2}$$

The constraints of the model can now be formulated. First off, we want to map every processing function in the graph to the DSPs on the grid by the constraints

$$\sum_{(i,j) \in \mathcal{I} \times \mathcal{J}} y_{m,n,(i,j)} = 1, \qquad n \in \mathcal{N}_m, \quad m \in \mathcal{M}, \tag{4.3}$$

which force all functions to be mapped.

As mentioned in Section 2.1, the function sequence framework allows for multiple functions $f_{m,n}$ to represent the same processing function in the connection nodes of the graph. For this reason there is need for constraints on functions in these nodes. Once again, it is assumed that the sequences are constructed such that only initial and final functions can be common, which yields the constraints

$$y_{m,N_m,(i,j)} - y_{\tilde{m},1,(i,j)} = 0, \qquad (m,\tilde{m}) \in \mathcal{M}_t^N \times \mathcal{M}_t^1, \quad t \in \mathcal{T}, \quad (i,j) \in \mathcal{I} \times \mathcal{J}, \tag{4.4}$$

where $\mathcal{M}_t^N \times \mathcal{M}_t^1$ is the product of sets of sequences ending and starting in the connection node $t \in \mathcal{T}$; $\mathcal{M}_t^N \subseteq \mathcal{M}$ and $\mathcal{M}_t^1 \subseteq \mathcal{M}$, respectively. That is, for a fixed t we have for $(m,\tilde{m}) \in \mathcal{M}_t^N \times \mathcal{M}_t^1$ that $f_{m,N_m} = f_{\tilde{m},1}$. The constraints (4.4) also imply that sequences having initial or final functions in common, will be placed on the same position on the grid, i.e., if $m, \hat{m} \in \mathcal{M}_t^N$ then $f_{m,N_m} = f_{\hat{m},N_{\hat{m}}}$, and if $m, \hat{m} \in \mathcal{M}_t^1$ then $f_{m,1} = f_{\hat{m},1}$. If the input (or the output) is represented by a connection node $t \in \mathcal{T}$ in the processing graph, then the corresponding sequence $m \in \mathcal{M}_{\text{input}} \subset \mathcal{M}_t^N$ (or $m \in \mathcal{M}_{\text{output}} \subset \mathcal{M}_t^1$) is defined to be a sequence with only the input (or output) function, which implies that $N_m = 1$.

In the case considered in this work, a satisfying mapping should not assign multiple processing functions to the same DSP. Hence, constraints to restrict the number of functions assigned to a postion (i,j) are needed. Thus, for all $(i,j) \in \mathcal{I} \times \mathcal{J}$,

$$\begin{aligned}&\sum_{t\in\mathcal{T}} \frac{1}{|\mathcal{M}_t^1| + |\mathcal{M}_t^N|}\left(\sum_{m\in\mathcal{M}_t^1} y_{m,1,(i,j)} + \sum_{m\in\mathcal{M}_t^N} y_{m,N_m,(i,j)}\right)\\ &+ \sum_{m\in\mathcal{M}} \sum_{n\in\mathcal{N}_m\setminus\{1,N_m\}} y_{m,n,(i,j)} + \sum_{m\in\mathcal{M}_{\text{input}}^+} y_{m,1,(i,j)} + \sum_{m\in\mathcal{M}_{\text{output}}^+} y_{m,N_m,(i,j)} \leq 1,\end{aligned} \tag{4.5}$$

where $\mathcal{M}_{\text{input}}^+ \subseteq \mathcal{M}_{\text{input}}$ and $\mathcal{M}_{\text{output}}^+ \subseteq \mathcal{M}_{\text{output}}$ are the sets of input/output sequences containing more than just the input/output functions. The sets $\mathcal{M}_{\text{input}}^+$ and $\mathcal{M}_{\text{output}}^+$ are needed when the input, or the output, is represented by a connecting node in the processing graph; otherwise the constraints (4.5) and (4.4) would contradict each other. The constraints (4.5) make sure that we do not assign more than one processing function on each position. If the function is common to multiple sequences they allow the corresponding y variables to take the value 1 on the same position. If the function is not common to multiple sequences then only one y variable can take the value 1.

Assuming that one can interpret input/output as functions to be assigned to DSPs on the grid, they should be assigned to positions on the top and bottom rows, respectively. The constraints are formulated as

$$\sum_{j\in\mathcal{J}} y_{m,1,(1,j)} = 1, \qquad m \in \mathcal{M}_{\text{input}}, \tag{4.6}$$

and

$$\sum_{j\in\mathcal{J}} y_{m,N_m,(I,j)} = 1, \qquad m \in \mathcal{M}_{\text{output}}. \tag{4.7}$$

There is a maximum number of links that can be used between neighouring DSPs, so a constraint to limit function coupling from exceeding this number is needed. Letting $L \in \mathbb{Z}_+$ be the link capacity, these constraints are then formulated as

$$\sum_{m\in\mathcal{M}} \sum_{n\in\mathcal{N}_m} \left(x_{m,n,(i,j),(q,r)} + x_{m,n,(q,r),(i,j)}\right) \leq L, \qquad (q,r) \in \mathcal{V}(i,j),\ (i,j) \in \mathcal{I} \times \mathcal{J}. \tag{4.8}$$

The coupling of function sequences can be interpreted as a commodity flow between DSPs. Each sequence corresponds to a commodity, so it is reminiscent to a multicommodity flow constraint. When determining a path between a pair of functions, it is therefore necessary to retain continuity in the flow, so that no "commodity" flow $x_{m,n,(i,j),(.,.)}$ is lost in any of the positions (i,j). The constraints to conserve multicommodity flow in the positions where functions are assigned read as

$$\sum_{(k,l)\in\mathcal{V}(i,j)} \Big(x_{m,n,(i,j),(k,l)} - x_{m,n,(k,l),(i,j)}\Big) = y_{m,n,(i,j)} - y_{m,n+1,(i,j)}, \tag{4.9}$$

for $(i,j) \in \mathcal{I} \times \mathcal{J}$ and $n \in \mathcal{N}_m \setminus \{N_m\}$, $m \in \mathcal{M}$. For an intermediate DSP position (i,j), there will be no functions assigned (i.e., $y_{m,n,(i,j)} = 0$) and the in- and out-flows to neighbouring DSPs are therefore equal. If a DSP position (i,j) is the source of data transmission between a pair of functions, i.e., a function $f_{m,n}$ has been assigned to it, then there is a unit surplus of couplings to the neighbouring DSPs. For sinks $f_{m,n+1}$, the converse holds. The constraints (4.9) imply implicitly that the coupling variables x, interpreted as commodity flow variables, are binary. Hence, when implementing the model, it suffices to restrict the variables $x_{m,n,(i,j),(k,l)}$, defined in (4.1), to the interval $[0,1]$, making the model type MBLP.

4.2 Dantzig-Wolfe Formulation

In making the decomposition into sub- and master problem, it is common to analyze if there is any apparent or natural partitioning of the constraints. For example, if the model is similar to some well-studied problem, with a few additional simple constraints; see [4, Section 1]. It could then be beneficial to form a more extensive master problem with the complicating constraints from the studied problem, and put the simple constraints in the subproblem. However, it is hard to know beforehand which is the best decomposition. For the above MBLP model, two different decompositions will be made, both having an intuitive interpretation.

4.2.1 Decomposition with Separated Sequences - Block Structure

The constraints in (4.3), (4.6), (4.7), and (4.9), all contains sums over positions, but not over different sequences. Hence a block-diagonal structure can be achieved in the matrix D, by placing these constraints in the subproblem. Thereby, a subproblem is formed for each sequence $m \in \mathcal{M}$. The remaining constraints are left to the master problem. Roughly speaking, the process of column generation would then amount to assigning separated sequences with corresponding links individually, via the subproblems, and then connect them through the RMP.

4.2.1.1 Restricted Master Problem for Separated Sequences

Denote the feasible solutions from the subproblems by $(y_p^\top, x_p^\top)$, where $y_p^\top = (y_p^m)_{m\in\mathcal{M}}$ and $x_p^\top = (x_p^m)_{m\in\mathcal{M}}$, and variables by λ_p^m, $p \in \mathcal{P}'_m$, $m \in \mathcal{M}$. The indices m and p then indicate subproblem and column, respectively. Having a block structure in the subproblem, the solutions to each subproblem m generate "part-columns" defined through y_p^m and x_p^m. Elements in the solutions will be denoted $y_{p,m,n,(i,j)}$ and $x_{p,m,n,(i,j),(k,l)}$.

Formulate the constraints of the RMP using (4.4), (4.5), and (4.8). The connection node constraints (4.4) become

$$\sum_{p\in\mathcal{P}'_m} \left(\lambda_p^m \cdot y_{p,m,N_m,(i,j)} - \lambda_p^{\tilde{m}} \cdot y_{p,\tilde{m},1,(i,j),}\right) = 0, \qquad \begin{array}{l}(m,\tilde{m}) \in \mathcal{M}_t^N \times \mathcal{M}_t^1,\\ (i,j) \in \mathcal{I}\times\mathcal{J},\ t \in \mathcal{T}.\end{array} \tag{4.10a}$$

The function assignment restriction constraints (4.5), become, for all $(i,j) \in \mathcal{I}\times\mathcal{J}$,

$$\begin{aligned}&\sum_{t\in\mathcal{T}} \frac{1}{|\mathcal{M}_t^1| + |\mathcal{M}_t^N|}\left(\sum_{m\in\mathcal{M}_t^1}\sum_{p\in\mathcal{P}'_m} \lambda_p^m \cdot y_{p,m,1,(i,j)} + \sum_{m\in\mathcal{M}_t^N}\sum_{p\in\mathcal{P}'_m} \lambda_p^m \cdot y_{p,m,N_m,(i,j)}\right) + \\ &\sum_{m\in\mathcal{M}}\sum_{p\in\mathcal{P}'_m}\sum_{n\in\mathcal{N}_m\setminus\{1,N_m\}} \lambda_p^m \cdot y_{p,m,n,(i,j)} + \\ &\sum_{m\in\mathcal{M}^+_{\text{input}}}\sum_{p\in\mathcal{P}'_m} \lambda_p^m \cdot y_{p,m,1,(i,j)} + \sum_{m\in\mathcal{M}^+_{\text{output}}}\sum_{p\in\mathcal{P}'_m} \lambda_p^m \cdot y_{p,m,N_m,(i,j)} \le 1.\end{aligned} \tag{4.10b}$$

The link capacity constraints (4.8) are formulated for $(q,r) \in \mathcal{V}(i,j)$, $(i,j) \in \mathcal{I}\times\mathcal{J}$, as

$$\sum_{m\in\mathcal{M}}\sum_{p\in\mathcal{P}'_m}\sum_{n\in\mathcal{N}_m} \lambda_p^m \cdot \left(x_{p,m,n,(i,j),(q,r)} + x_{p,m,n,(q,r),(i,j)}\right) \le L, \tag{4.10c}$$

A Dantzig-Wolfe reformulation with a block-diagonal structure also adds a convexity constraint for each subproblem. In this case there is a subproblem for each function sequence m and thus we add the constraints

$$\sum_{p\in\mathcal{P}'_m} \lambda_p^m = 1, \qquad \lambda_p^m \ge 0,\, p \in \mathcal{P}'_m, \quad m \in \mathcal{M}. \tag{4.10d}$$

Finally, the objective function of the RMP reads

$$z_{\text{RMP}} = \sum_{m\in\mathcal{M}}\sum_{p\in\mathcal{P}'_m}\sum_{n\in\mathcal{N}_m}\sum_{(i,j)\in\mathcal{I}\times\mathcal{J}}\sum_{(q,r)\in\mathcal{V}(i,j)} \lambda_p^m \cdot x_{p,m,n,(i,j),(q,r)}. \tag{4.10e}$$

4.2.1.2 Subproblems for Separated Sequences

The subproblems are defined in the same variable domain as the complete model in Section 4.1, with constraints as in (4.3), (4.6), (4.7) and (4.9).

Again, the constraints are for each subproblem $m \in \mathcal{M}$,

$$\sum_{(i,j)\in\mathcal{I}\times\mathcal{J}} y_{m,n,(i,j)} = 1, \quad n \in \mathcal{N}_m.$$

For $m \in \mathcal{M}_{\text{input}}$ and $m \in \mathcal{M}_{\text{output}}$ the constraints are

$$\sum_{j\in\mathcal{J}} y_{m,1,(1,j)} = 1,$$

and

$$\sum_{j\in\mathcal{J}} y_{m,N_m,(|\mathcal{I}|,j)} = 1,$$

respectively, and for $(i,j) \in \mathcal{I} \times \mathcal{J}$ and $n \in \mathcal{N}_m \setminus \{N_m\}$,

$$\sum_{(k,l)\in\mathcal{V}(i,j)} \left(x_{m,n,(i,j),(k,l)} - x_{m,n,(k,l),(i,j)}\right) = y_{m,n,(i,j)} - y_{m,n+1,(i,j)}.$$

In the subproblems we update the objective by penalizing the violation of constraints in the RMP, using the dual variable values computed from the solution to the RMP. Since each subproblem is associated with a sequence m, the terms of a subproblems objective will be connected to constraints and dual variables involving that particular sequence. Therefore, the terms necessary to formulate the objective of all subproblems will be stated below.

Firstly, the unpenalized objective function is the same as in the original formulation (4.2). For a specific subproblem m the share of the objective function is therefore

$$z_m := \sum_{n\in\mathcal{N}_m} \sum_{(i,j)\in\mathcal{I}\times\mathcal{J}} \sum_{(q,r)\in\mathcal{V}(i,j)} x_{m,n,(i,j),(q,r)}. \tag{4.11a}$$

To penalize the violation of constraints in the RMP, dual variables corresponding to the constraints are used. Denote dual variables corresponding to (4.10a), (4.10b), (4.10c), (4.10d) with $\alpha_{t,(m,\tilde{m}),(i,j)}$, $\beta_{(i,j)}$, $\gamma_{(i,j),(q,r)}$, δ_m, respectively. Beginning with the penalizing of (4.10a), there will be terms for sequences $m \in \mathcal{M}_t^N$ and $\tilde{m} \in \mathcal{M}_t^1$, for all $t \in \mathcal{T}$. For the corresponding constraints in (4.10a), optimal dual variable values $\hat{\alpha}_{t,(m,\tilde{m}),(i,j)}$, will be computed. Thus, for sequences $m \in \mathcal{M}_t^N$, $t \in \mathcal{T}$, add the term

$$-\sum_{m\in\mathcal{M}_t^N} \sum_{\tilde{m}\in\mathcal{M}_t^1} \sum_{(i,j)\in\mathcal{I}\times\mathcal{J}} \hat{\alpha}_{t,(m,\tilde{m}),(i,j)} \cdot y_{m,N_m,(i,j)} \tag{4.11b}$$

to the objective. Analogously, for sequences $\tilde{m} \in \mathcal{M}_t^1$, $t \in \mathcal{T}$, add the term

$$\sum_{\tilde{m}\in\mathcal{M}_t^1} \sum_{m\in\mathcal{M}_t^N} \sum_{(i,j)\in\mathcal{I}\times\mathcal{J}} \hat{\alpha}_{t,(m,\tilde{m}),(i,j)} \cdot y_{\tilde{m},1,(i,j)}. \tag{4.11c}$$

Penalizing the assignment restriction constraint (4.10b), there are several terms to be added. Here, the optimal dual variable value is $\hat{\beta}_{(i,j)}$. For sequence $m \in \mathcal{M}_t^N$, add the term

$$-\sum_{t\in\mathcal{T}}\sum_{(i,j)\in\mathcal{I}\times\mathcal{J}} \frac{1}{|\mathcal{M}_t^1| + |\mathcal{M}_t^N|}\hat{\beta}_{(i,j)} \cdot y_{m,N_m,(i,j)}. \tag{4.11d}$$

For sequence $m \in \mathcal{M}_t^1$, add the term

$$-\sum_{t\in\mathcal{T}}\sum_{(i,j)\in\mathcal{I}\times\mathcal{J}} \frac{1}{|\mathcal{M}_t^1| + |\mathcal{M}_t^N|}\hat{\beta}_{(i,j)} \cdot y_{m,1,(i,j)}. \tag{4.11e}$$

For $m \in \mathcal{M}$, add the term

$$-\sum_{(i,j)\in\mathcal{I}\times\mathcal{J}}\ \sum_{n\in\mathcal{N}_m\setminus\{1,N_m\}} \hat{\beta}_{(i,j)} \cdot y_{m,n,(i,j)}. \tag{4.11f}$$

For $m \in \mathcal{M}_{\text{input}}^+$ add the term

$$-\sum_{(i,j)\in\mathcal{I}\times\mathcal{J}} \hat{\beta}_{(i,j)} \cdot y_{m,1,(i,j)}. \tag{4.11g}$$

For $m \in \mathcal{M}_{\text{output}}^+$ add the term

$$-\sum_{(i,j)\in\mathcal{I}\times\mathcal{J}} \hat{\beta}_{(i,j)} \cdot y_{m,N_m,(i,j)}. \tag{4.11h}$$

To penalize the link capacity constraint (4.10c) with dual variables $\gamma_{(i,j),(q,r)}$, add for $m \in \mathcal{M}$, the term

$$-\sum_{(i,j)\in\mathcal{I}\times\mathcal{J}}\ \sum_{(q,r)\in\mathcal{V}(i,j)} \hat{\gamma}_{(i,j),(q,r)} \cdot \left(\sum_{n\in\mathcal{N}_m}\left(x_{m,n,(i,j),(q,r)} + x_{m,n,(q,r),(i,j)}\right)\right) \tag{4.11i}$$

to the objective.

Finally, to penalize the convexity constraints (4.10d), add $-\hat{\delta}_m$ to the objective of subproblem $m \in \mathcal{M}$.

The optimal objective values z_m from the M subproblems are reduced costs for variables λ_p^m which are to be added to the RMP. These minimized reduced costs will be denoted $\tilde{c}_p^m$ and will also be used to compute lower bounds on the MP in each iteration, through the inequality

$$\hat{z}_{\text{RMP}} + \sum_{m\in\mathcal{M}} \tilde{c}_p^m \leq \hat{z}_{\text{MP}}, \tag{4.12}$$

for the current generated full column $p \in \mathcal{P}'$.

4.2.2 Decomposition with Connected Sequences

The second decomposition of the original formulation in Section 4.1 is similar to the first. The difference is that the connection node constraints (4.4) are moved to the subproblem, making it harder to solve. A consequence of this decomposition is that the block-diagonal structure in the matrix D is lost, and hence there is only one subproblem. Thus, in this decomposition the column generation amounts to assigning the entire processing graph via the subproblem, and the RMP enforces the assignment restriction (4.5) and link capacity (4.8).

4.2.2.1 Restricted Master Problem for Connected Sequences

In this decomposition, denote the feasible solutions to the subproblem, defining the full columns, by $(y_p^\top, x_p^\top)$, and variables by λ_p, where $p \in \mathcal{P}'$. Similar to (4.10b) and (4.10c) above, the function assignment constraints are now

$$\begin{aligned}&\sum_{p\in\mathcal{P}'} \lambda_p \cdot \Bigg(\sum_{t\in\mathcal{T}} \frac{1}{|\mathcal{M}_t^1| + |\mathcal{M}_t^N|} \Bigg[\sum_{m\in\mathcal{M}_t^1} y_{p,m,1,(i,j)} + \sum_{m\in\mathcal{M}_t^N} y_{p,m,N_m,(i,j)} \Bigg] + \\ &\sum_{m\in\mathcal{M}} \sum_{n\in\mathcal{N}_m\setminus\{1,N_m\}} y_{p,m,n,(i,j)} + \sum_{m\in\mathcal{M}_{\text{input}}^+} y_{p,m,1,(i,j)} + \sum_{m\in\mathcal{M}_{\text{output}}^+} y_{p,m,N_m,(i,j)} \Bigg) \leq 1,\end{aligned} \tag{4.13a}$$

for all $(i,j) \in \mathcal{I} \times \mathcal{J}$, and the link capacity constraints are formulated for $(q,r) \in \mathcal{V}(i,j)$, $(i,j) \in \mathcal{I} \times \mathcal{J}$, as

$$\sum_{p\in\mathcal{P}'} \lambda_p \cdot \sum_{m\in\mathcal{M}} \sum_{n\in\mathcal{N}_m} \left(x_{p,m,n,(i,j),(q,r)} + x_{p,m,n,(q,r),(i,j)} \right) \leq L. \tag{4.13b}$$

Since the block-diagonal structure is lost, there is now only one convexity constraint in the RMP,

$$\sum_{p\in\mathcal{P}'} \lambda_p = 1. \tag{4.13c}$$

The objective function of the RMP now reads,

$$z_{\text{RMP}} = \sum_{p\in\mathcal{P}'} \lambda_p \cdot \sum_{m\in\mathcal{M}} \sum_{n\in\mathcal{N}_m} \sum_{(i,j)\in\mathcal{I}\times\mathcal{J}} \sum_{(q,r)\in\mathcal{V}(i,j)} x_{p,m,n,(i,j),(q,r)}. \tag{4.13d}$$

4.2.2.2 Subproblem for Connected Sequences

In this decomposition there is only one subproblem to solve. The constrains in the single subproblem are the same as above, (4.3), (4.6), (4.7) and (4.9), with the addition of the connection node constraints in (4.4). The constraints are thus

$$\sum_{(i,j)\in\mathcal{I}\times\mathcal{J}} y_{m,n,(i,j)} = 1, \quad n\in\mathcal{N}_m,\ m\in\mathcal{M},$$

$$y_{m,N_m,(i,j)} - y_{\tilde{m},1,(i,j)}, = 0, \quad (m,\tilde{m})\in\mathcal{M}_t^N\times\mathcal{M}_t^1, \quad t\in\mathcal{T}, \quad (i,j)\in\mathcal{I}\times\mathcal{J},$$

$$\sum_{j\in\mathcal{J}} y_{m,1,(1,j)} = 1, \quad m\in\mathcal{M}_{\text{input}},$$

$$\sum_{j\in\mathcal{J}} y_{m,N_m,(|\mathcal{I}|,j)} = 1, \quad m\in\mathcal{M}_{\text{output}},$$

and for $(i,j)\in\mathcal{I}\times\mathcal{J}$ and $n\in\mathcal{N}_m\setminus\{N_m\}$, $m\in\mathcal{M}$,

$$\sum_{(k,l)\in\mathcal{V}(i,j)} \Big(x_{m,n,(i,j),(k,l)} - x_{m,n,(k,l),(i,j)}\Big) = y_{m,n,(i,j)} - y_{m,n+1,(i,j)}.$$

The single unpenalized objective function is now

$$z = \sum_{m\in\mathcal{M}}\sum_{n\in\mathcal{N}_m}\sum_{(i,j)\in\mathcal{I}\times\mathcal{J}}\sum_{(q,r)\in\mathcal{V}(i,j)} x_{m,n,(i,j),(q,r)}. \tag{4.14}$$

For the penalization of violation of the constraints (4.13a) and (4.13b) in the RMP, the terms (4.11d)–(4.11i) can be used. All terms should be added to the objective function. To penalize the convexity constraint (4.13c) the dual variable δ is used, and the term $-\hat{\delta}$ with the optimal dual variable value, should be added to the objective.

4.2.3 Implementation Strategies

In this section, a couple of implementation strategies for the column generation method are briefly described. For generality, the notation introduced in Section 3.8 will be used. First, the approach used to initiate the RMP, will be covered. The concept of "Hot Start" (also referred to as warm start) is then introduced, as a method of improving the running time of the column generation algorithm. Finally, a procedure to improve the convergence rate of the method, is given.

4.2.3.1 Initialization of the RMP

The first iteration step of the column generation algorithm is to optimize the RMP to compute dual variable values. For this an initial column $\left(c^\top x_p, (Ax_p)^\top\right)$ is needed. It suffices that the solution x_p is feasible in the subproblem, but to guarantee feasible solutions when the column generation algorithm finalizes, an initial solution feasible in the original formulation, can be used to define the first column. This solution can be found by means of branch-and-bound and terminating the algorithm when the first feasible solution is found. The RMP can then be initiated using an artificial convexity variable $\lambda_0 \geq 0$, with a high cost M in the objective function [4, Section 1.2], forming the initial objective function

$$z_{\text{RMP}} := M\lambda_0.$$

For a block-diagonal structure the initial objective becomes

$$z_{\text{RMP}} := M \sum_{m \in \mathcal{M}} \lambda_0^m.$$

Optimizing the RMP now yields optimal values for a set of dual variables that can be used in the subproblem, and the column generation can commence.

4.2.3.2 Hot Start

The RMP is a linear program which can be solved by the simplex method. Thus, when an optimal solution is found, a corresponding optimal basis B has also been computed; see Section 3.1.1. With the current basis comes a set of basic and non-basic convexity variables λ_p^B and λ_p^N, respectively, in the partitioning

$$\lambda_p = \begin{pmatrix} \lambda_p^B \\ \lambda_p^N \end{pmatrix}, \qquad p \in \mathcal{P}'.$$

When using the simplex method to optimize the next RMP instance (i.e., the instance of the RMP in the next column generation iteration), the basis status of the previous optimal variable values can be used as an initial BFS in the optimization of the new RMP; see [8], [9]. Since it is possible that the basis status for many of the variables in the previous iteration remains unchanged when a new variable is added to the RMP, this procedure can speed up the search for the next optimal BFS. It is therefore a good idea to save the basis B at each iteration of the column generation algorithm. For large problem instances, however, the number of entries in B can be huge and the computation of the inverse B^{-1}, which is needed for the computation of reduced costs in the simplex algorithm, is expensive. To further speed up the running time of the column generation, it is therefore essential that the basis inverse is saved in each iteration.

4.2.3.3 Suboptimal Columns

The minimization of the subproblem amounts to finding a single column $\left(c^\top x_p, (Ax_p)^\top\right)$, $p \in \mathcal{P} \setminus \mathcal{P}'$, with the most negative reduced cost $\tilde{c}_p = (c^\top - \pi^\top A)x_p - \pi_0$ in the current optimal basis. However, there might exist other columns with negative reduced costs, which are not minimal. Since the subproblem is solved by branch-and-bound, these non-minimal negative reduced costs can be found as an intermediate solution in the BnB-tree. These suboptimal columns $\left(c^\top x_{\tilde{p}}, (Ax_{\tilde{p}})^\top\right)$, $\tilde{p} \in \mathcal{P} \setminus \mathcal{P}'$, can therefore be saved and added to the RMP, with a corresponding convexity variable $\lambda_{\tilde{p}}$, together with the optimal column. This procedure might improve the convergence speed of the column generation algorithm, since additional promising variables are added to the RMP in each iteration [4, Sec. 2.3].

For the case of a block-diagonal structure, each subproblem has potential suboptimal solutions in their respective branching tree. Since the subproblems can be differently hard to solve, the number of explored nodes in the BnB-tree might differ among the subproblems. Therefore, each subproblem may contribute with different numbers of suboptimal part-columns.

5 Tests and Results

Below follows tests and results on the three model formulations in Chapter 4. The original formulation in Section 4.1 was solved to optimality by means of branch-and-bound. In an effort to improve computational running time, approximately optimal solutions to the original formulation were found through the Dantzig-Wolfe reformulations in Sections 4.2.1–4.2.2, and column generation as presented in Section 3.8.

Softwares and strategies used to implement the methods are presented, together with specifications of hardware that was used to run the implementations. Further, the different problem instances are presented in terms of a processing graph and problem parameters. The results from the problem instances, regarding the convergence of the column generation algorithm, a comparison of resulting integer solutions and the computational running times, are then discussed.

5.1 Implementation and Hardware

To implement the methods used, the models were developed in the high-level modelling language *JuMP* (v.0.19) [10], [11], and the optimization software *Gurobi* (v.8.1) was used as a solver [12], [13]. The JuMP modelling language is a package in the *Julia* (v.1.1) programming language [14], [15], which was also used during code development.

Gurobi is an optimization solver which was used to solve the original formulation in Section 4.1, and for solving the subproblems and RMPs in the reformulations in Sections 4.2.1–4.2.2. For the original formulation and subproblems, the branch-and-bound solver was used, and for the RMPs the simplex solver was used. Gurobi utilizes a parallelization framework when possible, and distributes computational tasks on the available processing cores. Especially the branch-and-bound solver takes advantage of this. Hence, to compare running times for different problem instances, the CPU time was measured. For the reformulation with a block-diagonal structure, each subproblem is solved sequentially and Gurobi then parallelizes the computation of the current subproblem instance.

To initiate the RMPs, the original formulation was processed by Gurobis branch-and-bound solver until the first feasible solution was found. The cost of the artificial variable was chosen to $M = 1000$, in all implemented variants of the column generation. A form of hot start initiations was used for the RMPs, i.e., the basis status of the variables in the RMPs was saved after each iteration and the previous basis B was given as input to the simplex solver at the next iteration. However, the basis inverse B^{-1} still needed to be computed at each iteration. For some problem instances of the reformulations, intermediate solutions were saved in the branch-and-bound solver used for the subproblems. A specification of the maximal number of saved intermediate solutions per iteration $P_{\mathrm{s}}^{\max}$, was then given as input to the branch-and-bound solver. A tolerance level for the reduced costs were set to 1e-9, i.e., if $\tilde{c}_p > -1\text{e-}9$ (or if $\tilde{c}_p^m > -1\text{e-}9$, for all $m \in \mathcal{M}$, for the cases with a block-diagonal structure), the column generation was terminated.

The hardware used was an Intel Xeon CPU E5-2683 v3 @ 2.00GHz processor [16], with 125 GiB RAM. To create figures *MATLAB* (v.R2018b), was used.

5.2 Tests on the Task Mapping Problem

In this section the problem instances are presented together with results on the task mapping problem. Each problem instance is defined by the method used and a case graph, together with a specific problem parameter combination. The results presented regard convergence of the column generation algorithm, the integer solutions to the instances, and computational aspects of the methods.

5.2.1 Test Settings

The methods implemented were tested on a number of different problem instances, as specficied in Table 5.1. The processing graph, which functions was to be mapped to, were in all cases the case graph illustrated in Figure 5.1. Two grid cases were tested, specified by the parameters L, I and J. Using the case graph and the function sequence framework, the problem was specified in terms of the model (4.2)–(4.9), presented in Section 4.1, which could therein be used for the DW-reformulations presented in Sections 4.2.1–4.2.2. Each formulation corresponds to a certain implemented method. For the instances where column generation was used, the maximal number of suboptimal solutions found when solving the subproblems are specified by the number $P_{\mathrm{s}}^{\max}$.

The grid parameter values were tested according to the settings

- $L = 10$, $I = 8$, $J = 8$, and
- $L = 4$, $I = 4$, $J = 10$.

These settings were chosen in order to investigate the effect of decreasing the number of degrees of freedom for the task mapping. When decreasing the grid size and the link capacity, it is harder to find a feasible solution, and the number of optimal solutions are reduced. Effectively, the size of the feasible region is reduced by reducing the degrees of freedom.

The different methods used are branch-and-bound (Gurobi-BnB), and column generation with and without a block structure, i.e., CG-Block and CG, respectively. The instances solved by branch-and-bound were used as a benchmark for the column generation method, in terms of objective function value and computational running time. The column generation methods will be compared in terms of the RMPs convergence to the MP, resulting integer solutions to the corresponding IRMP and computational running times. How the process of saving suboptimal solutions in the subproblem affects the results, will also be investigated.

Instance name	Method	L	I	J	$P_\mathrm{s}^\mathrm{max}$
B-10-8-8-1	CG-Block	10	8	8	1
NB-10-8-8-1	CG	10	8	8	1
B-10-8-8-100	CG-Block	10	8	8	100
NB-10-8-8-100	CG	10	8	8	100
G-10-8-8	Gurobi-BnB	10	8	8	-
B-4-4-10-1	CG-Block	4	4	10	1
NB-4-4-10-1	CG	4	4	10	1
B-4-4-10-100	CG-Block	4	4	10	100
NB-4-4-10-100	CG	4	4	10	100
G-4-4-10	Gurobi-BnB	4	4	10	-

Table 5.1: Problem instances. An instance is defined by the solution method used and the problem parameters. Each method corresponds to one of the problem formulations, i.e., the original formulation (Gurobi-BnB), reformulation with a block structure (CG-Block) and reformulation without a block structure (CG). The parameters are the link capacity L, the grid size specified by I and J, and the maximum number of saved suboptimal columns per iteration in the column generation, $P_\mathrm{s}^\mathrm{max}$.

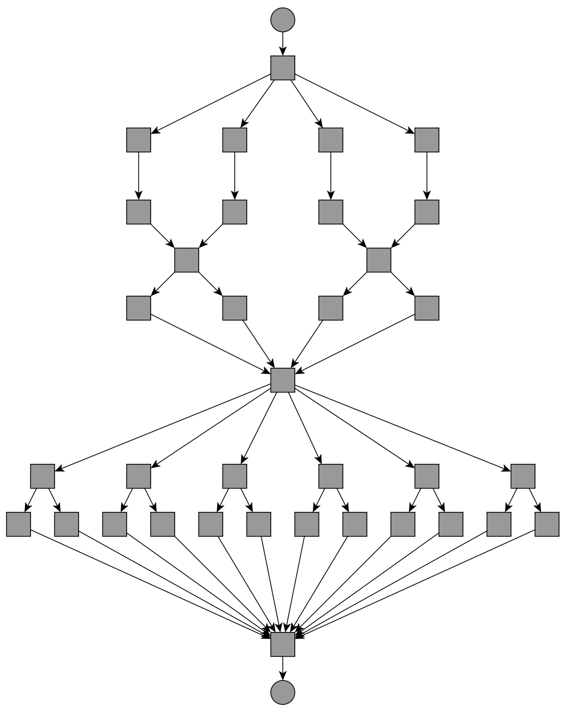
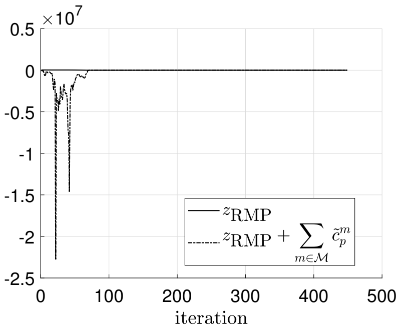


Figure 5.1: The case graph with processing functions and their dependencies. Input/Output are denoted by circles at the top/bottom, respectively. The graph consists of 37 processing functions (nodes), 28 function sequences and eleven connection nodes, i.e., $|\cup_{m\in\mathcal{M}}\mathcal{N}_m| = 37$, $M = 28$ and $|\mathcal{T}| = 11$.

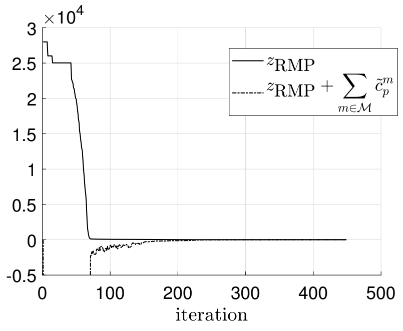
5.2.2 Convergence of the RMP

The convergence of the column generation algorithm is measured through the bounds on the optimal value of the (integer) master problem; see (3.37) and (3.38). In Figure 5.2 the convergence of the algorithm is shown for the 8×8 grid with a block-diagonal structure in the reformulation. In subfigures (a) and (d) it is evident that the upper bound converges much faster than the lower bound. It can also be seen that during most of the iterations, the bounds are relatively close, making it difficult to know if the algorithm is near termination. This could complicate the use of bounds as a termination criterion, since there is then a chance that the algorithm is terminated prematurely by a criterion using a large gap in the bounds and many iterations might be lost this way.

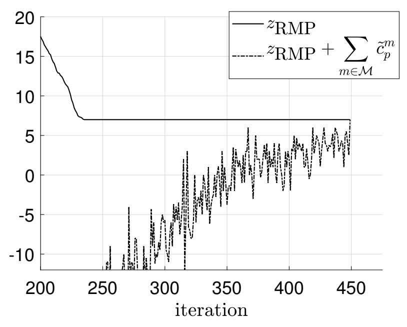
Initially, however, the magnitudes of the upper and lower bounds differ significantly. Since the lower bound is in the large negatives, and it is obvious that a feasible integer solution is positive, the lower bound is of no particular use as a lower bound to the MP (or the IMP), at least not until later iterations as seen in subfigures (c) and (f). The initially large negative value on the lower bound, is due to the cost of violating the constraints in the RMP. After a couple of iterations, sufficiently good columns have entered the RMP (in an LP sense), which do not violate the constraints to the same extent. Comparing (a) and (d), it is seen that using suboptimal columns in the RMP does not seem to improve the usefulness of the lower bound as a bound for the MP, but it reduced the number of iterations needed for termination. This could be explained by the fact that adding more columns in each iteration means that the set of extreme points, indexed by $p \in \mathcal{P}$, is being depleted at a faster rate.



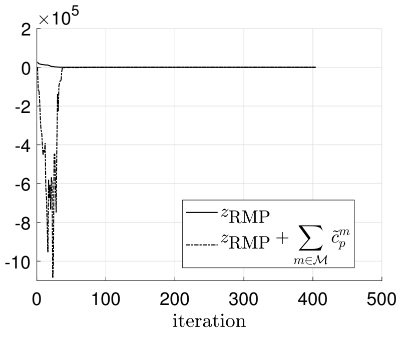
(a) Progress of the upper and lower bounds for the instance B-10-8-8-1.



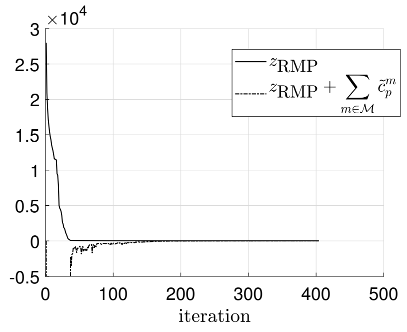
(b) A close-up of the upper bound progress for the instance B-10-8-8-1.



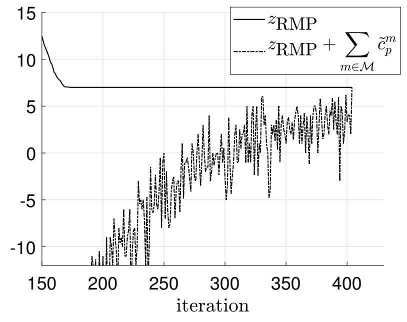
(c) A close-up of the final convergence of the bounds for the instance B-10-8-8-1.



(d) Progress of the upper and lower bounds for the instance B-10-8-8-100.



(e) A close-up of the upper bound progress for the instance B-10-8-8-100.

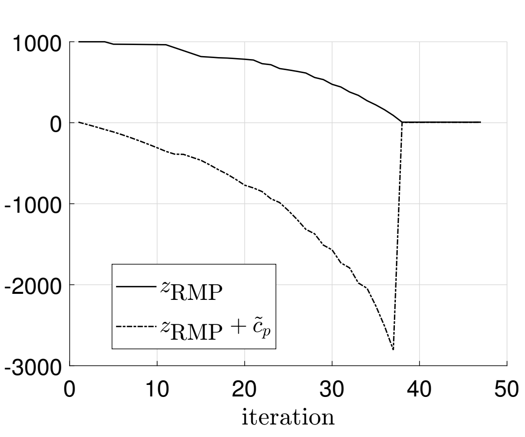


(f) A close-up of the final convergence of the bounds for the instance B-10-8-8-100.

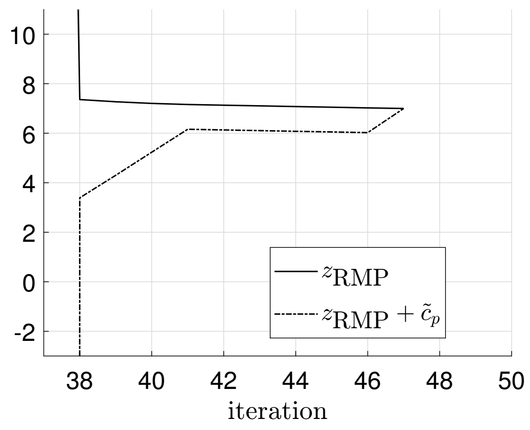
Figure 5.2: Upper and lower bounds on the optimal value of the MP using a block structure for the subproblems. The grid is specified by $L = 10$, $I = 8$ and $J = 8$. 'iteration' denotes the iteration in the column generation algorithm, and thereby also the current number of columns, corresponding to optimal solutions in the subproblems, in the RMP. The upper and lower bounds are given by the inequalities in (3.37) and (3.38).

In Figure 5.2, (b) and (e), magnifications of the upper bounds are shown. Also in this scale the bounds are relatively close for a larger part of the algorithm. The initially large positive value of the upper bound is explained by the initial high cost M, for the artificial variables λ_0^m. After a couple of iterations this cost will be entirely dimished, since newer columns will be cheaper in the objective. In (b) and (e) it becomes clear that the upper and lower bounds start to close in on the true value of $\hat{z}_{MP}$ (the bounds are confined to the interval $[-5, 5] \times 1000$), around the same iteration, ca $P = 75$ for (b) and $P = 40$ for (e). However, the upper bound reaches the final value faster, as seen in both (c) and (f).

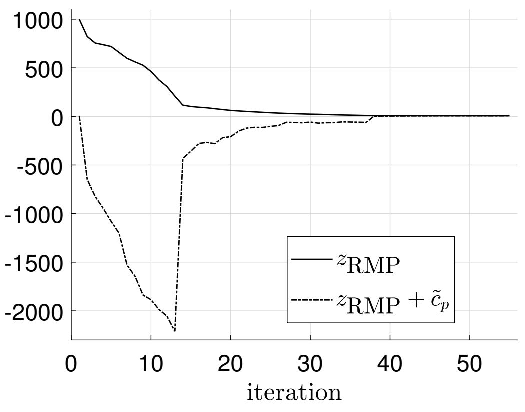
Figure 5.3 shows the convergence for the 8×8 grid for the case without a block structure. Compared to using a block structure, the convergence is faster, in terms of iterations, and



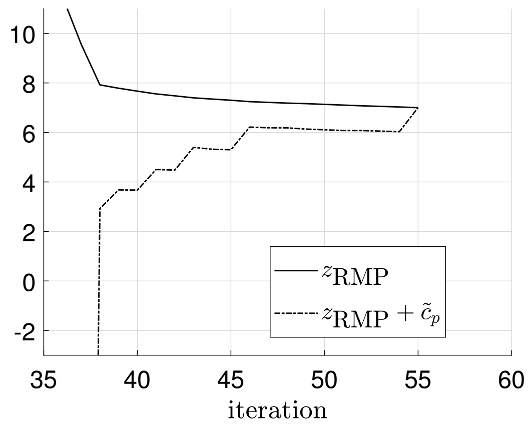
(a) Progress of the upper and lower bounds for the instance NB-10-8-8-1.



(b) A close-up of the final convergence of the bounds for the instance NB-10-8-8-1.



(c) Progress of the upper and lower bounds for the instance NB-10-8-8-100.



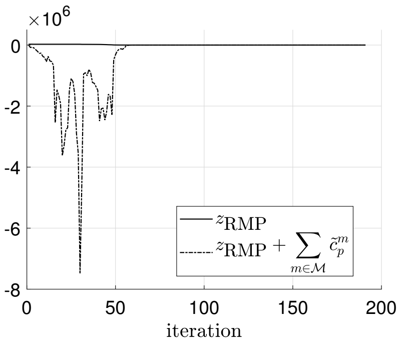
(d) A close-up of the final convergence of the bounds for the instance NB-10-8-8-100.

Figure 5.3: Upper and lower bounds on the optimal value of the MP without using a block structure for the subproblems. The grid is specified by $L = 10$, $I = 8$ and $J = 8$. 'iteration' denotes the iteration in the column generation algorithm, and thereby also the current number of columns, corresponding to optimal solutions in the subproblems, in the RMP. The upper and lower bounds are given by the inequalities in (3.37).

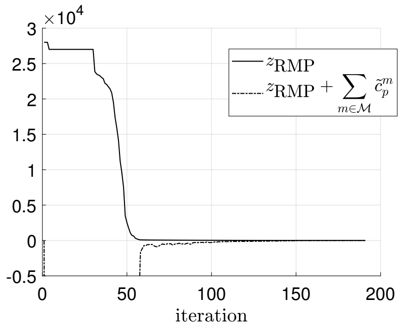
the upper and lower bounds are more similar in size. The faster convergence, could be explained by the fact that the subproblems are now harder to solve, since the constraints (4.4) were transfered to them. Thus, there are not as many constraints to comply with in the RMP, making the number of columns with negative reduced cost smaller. For this case the bounds provide a better indication of when the algorithm is near termination. When the bounds are relatively near each other, i.e., the bounds are confined to the interval $[-5, 10]$, the algorithm is about to terminate, as made clear by subfigures (b) and (d). These subfigures show that there are only a few iterations, ca 10 in (b) and 15 in (d), where the bounds are this close to each other, whereas for the case with a block structure in Figure 5.2, the corresponding numbers are ca 120 for (c) and 130 for (f). Comparing (a) and (c) in Figure 5.3, it is clear that saving suboptimal columns from the subproblem solution process did not improve the convergence rate of the algorithm. In fact, the number of optimal columns from the subproblems needed in the RMP, i.e., the number of iterations, increased when suboptimal columns from the subproblems were added to the RMP. However, it did decrease the upper bound on the optimal value of the MP, at an earlier stage of the algorithm.

Moving on, the results for the instance with the 4×10 grid and with a block structure, illustrated in Figure 5.4, is similar to that in Figure 5.2, in terms of bounds. The bounds are still relatively close to each other for a long duration of the algorithm, making it hard to distinguish whether or not the algorithm is close to terminating. However, the number of iterations where the bounds are confined to the interval $[-5, 10]$ is somewhat reduced as compared to 5.2, but so is the total number of iterations. As expected, the number of columns needed for the RMP to converge to the MP (again, the total number of iterations), was decreased for the 4×10 grid. It seems that adding suboptimal columns still increased the convergence rate in this case.

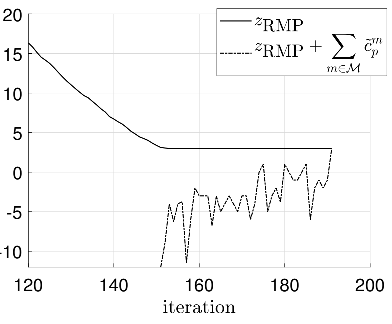
Finally, for the 4×10 grid without a block structure, see Figure 5.5, not much changed as compared to the 8×8 grid. The most distinct difference, as compared to Figure 5.3, is that the lower bound is lower in early iterations. To note is that the number of iterations needed for the convergence of the column generation algorithm is very similar to that of the 8×8 grid. The process of saving suboptimal columns did not improve the convergence in this case either. But more testing on grid sizes should be done in order to find out whether this is generally the case.



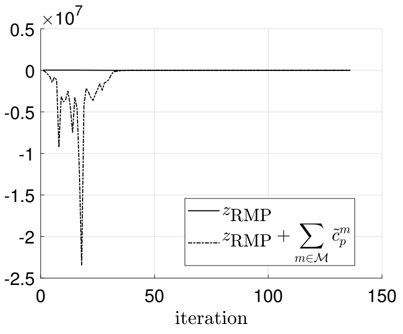
(a) Progress of the upper and lower bounds for the instance B-4-4-10-1.



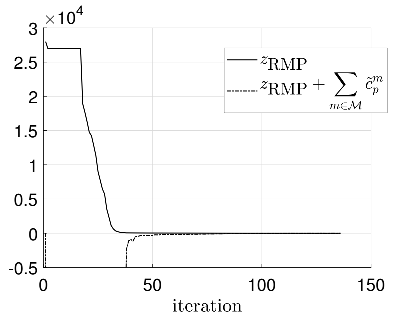
(b) A close-up of the upper bound progress for the instance B-4-4-10-1.



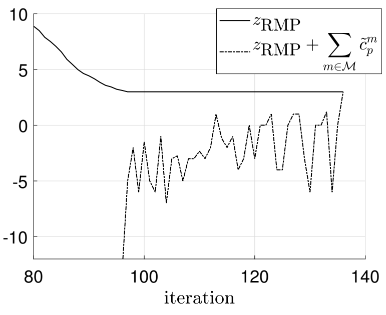
(c) A close-up of the final convergence of the bounds for the instance B-4-4-10-1.



(d) Progress of the upper and lower bounds for the instance B-4-4-10-100.

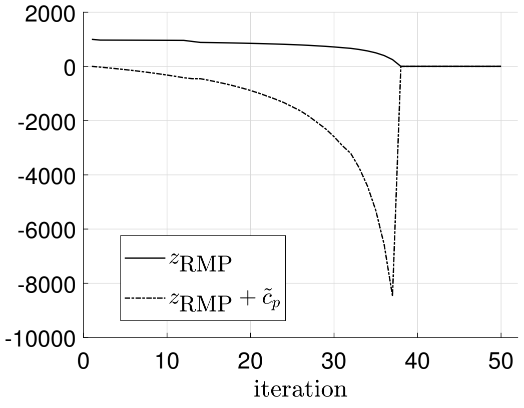


(e) A close-up of the upper bound progress for the instance B-4-4-10-100.

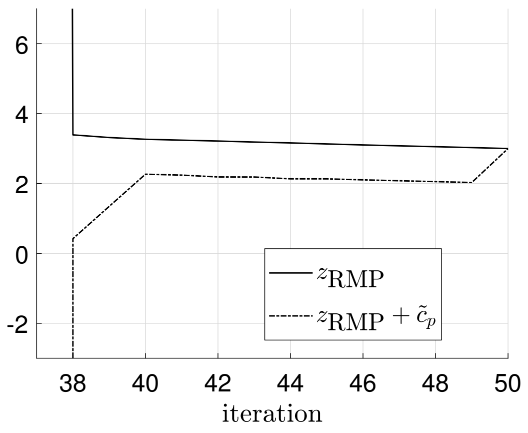


(f) A close-up of the final convergence of the bounds for the instance B-4-4-10-100.

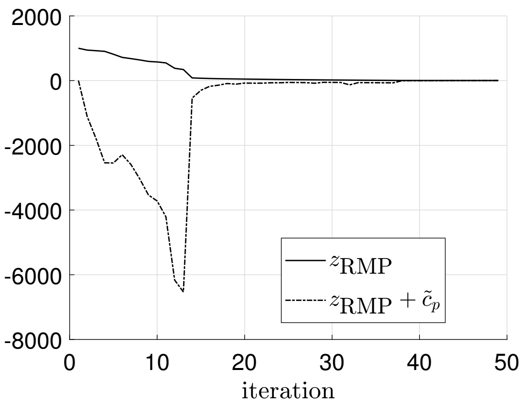
Figure 5.4: Upper and lower bounds on the optimal value of the MP using a block structure for the subproblems. The grid is specified by $L = 4$, $I = 4$ and $J = 10$. 'iteration' denotes the iteration in the column generation algorithm, and thereby also the current number of columns, corresponding to optimal solutions in the subproblems, in the RMP. The upper and lower bounds are given by the inequalities in (3.37) and (3.38).



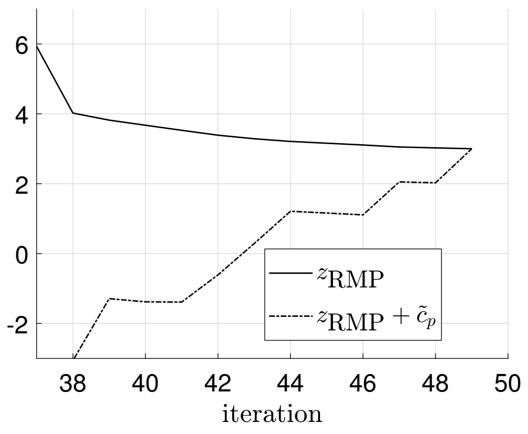
(a) Progress of the upper and lower bounds for the instance NB-4-4-10-1.



(b) A close-up of the final convergence of the bounds for the instance NB-4-4-10-1.



(c) Progress of the upper and lower bounds for the instance NB-4-4-10-100.



(d) A close-up of the final convergence of the bounds for the instance NB-4-4-10-100.

Figure 5.5: Upper and lower bounds on the optimal value of the MP without using a block structure for the subproblems. The grid is specified by $L = 4$, $I = 4$ and $J = 10$. 'iteration' denotes the iteration in the column generation algorithm, and thereby also the current number of columns, corresponding to optimal solutions in the subproblems, in the RMP. The upper and lower bounds are given by the inequalities in (3.37).

In Table 5.2, the termination data is shown for the column generation instances, together with the optimal objective values. It is evident that $\hat{z}_{\mathrm{RMP}}$ is not a good indication of the objective value of the corresponding binary solution, in any of the instances. With this information one can also conclude that the final RMP, did not fulfill integer restrictions on the columns. This was however expected and there might still exist feasible improvements to the initial feasible solution, among the columns generated.

The final number of generated optimal and suboptimal columns are also presented here. An indication of the complexity of the subproblems can be given by the quotient $\frac{P_{\mathrm{s}}}{P}$. For large values of the quotient, the subproblems are more complex. For a block structure, the number of suboptimal part-columns found can differ for each subproblem. Therefore the number of saved suboptimal part-columns is averaged over the subproblems for comparisons. This average represents the number of full suboptimal columns found. For both the large and small grid with a block structure formulation, the number of optimal columns needed for the RMP to converge was reduced when suboptimal columns were used. One can also see that for each optimal column found roughly 3 suboptimal columns are found, on average. This is the case for both the large and small grid.

Instance	$\hat{z}_{\mathrm{RMP}}$	$\hat{z}$	P	P_{s}	$\frac{P_{\mathrm{s}}}{P}$
B-10-8-8-1	7.00	82	449	-	-
B-10-8-8-100	7.00	82	404	1286.60	3.18
NB-10-8-8-1	7.00	82	47	-	-
NB-10-8-8-100	7.00	82	55	284	5.16
B-4-4-10-1	3.00	85	191	-	-
B-4-4-10-100	3.00	85	136	346.57	2.55
NB-4-4-10-1	3.00	85	50	-	-
NB-4-4-10-100	3.00	85	49	162	3.31

Table 5.2: Termination data for the column generation algorithm. $\hat{z}_{\mathrm{RMP}}$ is the objective value in the final iteration of the RMP, and thus also the objective value for the MP, i.e., $\hat{z}_{\mathrm{MP}} = \hat{z}_{\mathrm{RMP}}$. $\hat{z}$ is the optimal objective value of the problem instance. P denotes the number of columns in the final iteration of the RMP. P_{s} denotes the number of suboptimal solutions found in the subproblems. In the case of a block structure, P_{s} is the average number of suboptimal solutions found among the separated subproblems.

Without a block structure, the number of suboptimal subproblem solutions found per optimal subproblem solution differs for the large and small grid, from ca five to three. The quotient is higher than for the case with a block structure for both the large and small grid, which is to be expected since the subproblem is then more constrained. The larger decrease in the quotient between NB-10-8-8-100 and NB-4-4-10-100, could intuitively be explained by the fact that it should be harder to find an optimum for the mapping of the entire graph for the large grid, since there are more possibilities to choose from. When the grid is smaller and/or more restricted by the link capacity, the number of possibilities for the entire graph to be mapped, decreases heavily. In the case of a block structure individual, separated sequences are assigned. These sequences might not be as affected by the grid size, since each one of them represent relatively few nodes in the processing graph. Thus, the quotient should not decrease as significantly between grid sizes for the instances with a block structure, which seems to be the case here.

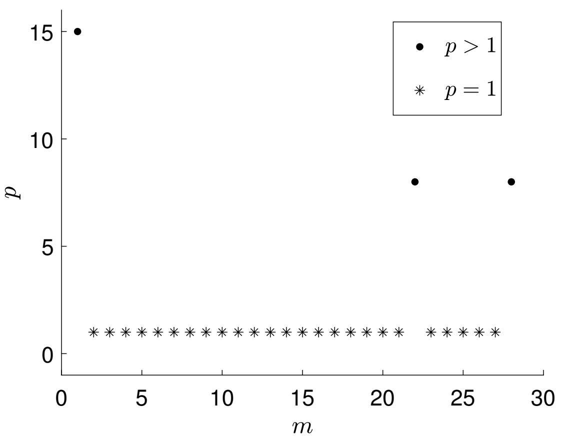
5.2.3 Binary solutions to the RMP

The final objective values to the binary restricted RMP are shown in Table 5.3. Only for the case of a block structure with the large grid, improvements to the initial feasible solution could be found. The improvement is increased slightly when suboptimal columns are used. However, the difference between optimal and approximately optimal objective value is significant. This shows that it is difficult to find separated sequences and link them together to create a feasibly assigned processing graph. But that it is possible, if enough part-columns are generated before termination of the algorithm. Without a block structure, no improvements were made. This suggests that assigning entire graphs in an LP sense is very different from employing binary constraints.

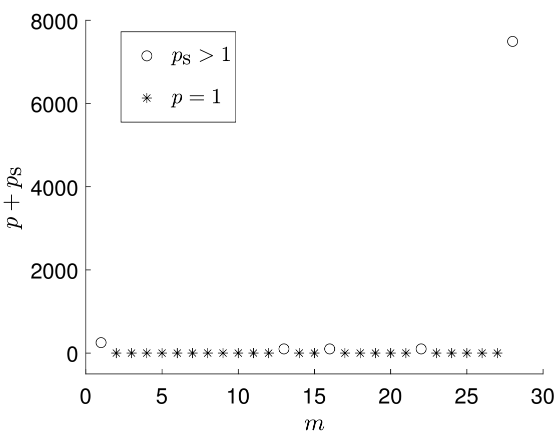
Instance	z_{init}	$\hat{z}_{\text{IRMP}}$	$\hat{z}$
B-10-8-8-1	455.0	447.0	82.0
B-10-8-8-100	455.0	443.0	82.0
NB-10-8-8-1	455.0	455.0	82.0
NB-10-8-8-100	455.0	455.0	82.0
B-4-4-10-1	127.0	127.0	85.0
B-4-4-10-100	127.0	127.0	85.0
NB-4-4-10-1	127.0	127.0	85.0
NB-4-4-10-100	127.0	127.0	85.0

Table 5.3: Final objective function values. z_{init} is the objective function value for the feasible solution used to initiate the RMP. $\hat{z}_{\text{IRMP}}$ is the optimal objective value of the final RMP when the convexity variables are restricted to binary values. $\hat{z}$ is the optimal objective value for the instance.

To investigate the cause of improvements in the instances B-8-8-10-1 and B-8-8-10-100, it is of interest to see which part-columns were chosen in the solution of the IRMP. Figure 5.6 shows for which subproblems an improving part-column was found. A comparison of (a) and (b) is consistent with the results in Table 5.3. In the case without suboptimal part-columns, three improving part-columns are found. When saving suboptimal part-columns, five improvements are found. Note that in (b), all improvements are found among the suboptimal part-columns, giving incentive for this implementation. In (b) one can also see that a suboptimal part-column found in the later stage of the algorithm was chosen,



(a) B-10-8-8-1.



(b) B-10-8-8-100.

Figure 5.6: Part-columns chosen in the IRMP. On the horizontal axis, m denotes the subproblem associated with an individual sequence. The characters p and p_s denote enumeration of optimal and suboptimal part-columns generated, respectively. Each dot represents the part-column for which the corresponding convexity variable $\lambda_p^m = 1$ in the IRMP. In (a), the enumerations on the vertical axis represent optimal part-columns only. In (b), the enumerations on the vertical axis represent both optimal and suboptimal part-columns. In (b), all improving part-columns chosen were from suboptimal solutions.

indicating that improvements can be found at any stage of the algorithm. It should be mentioned that an improving part-column chosen in the IRMP could be equivalent in cost to the corresponding initial part-column (in the initial feasible solution) in terms of the resulting connections mapped to the grid. In that case the apparently improving part-column is simply cheaper than the cost for the artificial variable corresponding to the initial part-column.

Common to both (a) and (b) in Figure 5.6, is that the first and last subproblems, i.e., $m = 1$, $m = 28$, generated improvements. In Figure 5.7 the distribution of the suboptimal part-columns found can be seen for the instances with a block structure. For both the small and large grid, it can be seen that the subproblems $m \in \{1, 28\}$ had significantly more intermediate solutions in the branch-and-bound tree, indicating that they were harder to solve. This is likely because of the additional constraints (4.6) and (4.7), concerning input and output, put on the corresponding sequences. The consequence is that, in the IRMP, there is a larger number of possibly improving combinations to choose from involving these two sequences, compared to the others. From this figure it also is clear that the average number of suboptimal columns found is not a particularly representative quantity. Most of the sequences generate only 1–2 suboptimal part-columns per iteration, indicating that the individual subproblems are very easy to solve. This suggests that restricting the task mapping problem through constraints on, e.g., the assignment of specific functions, could improve the resulting integer solution produced through column generation. The model

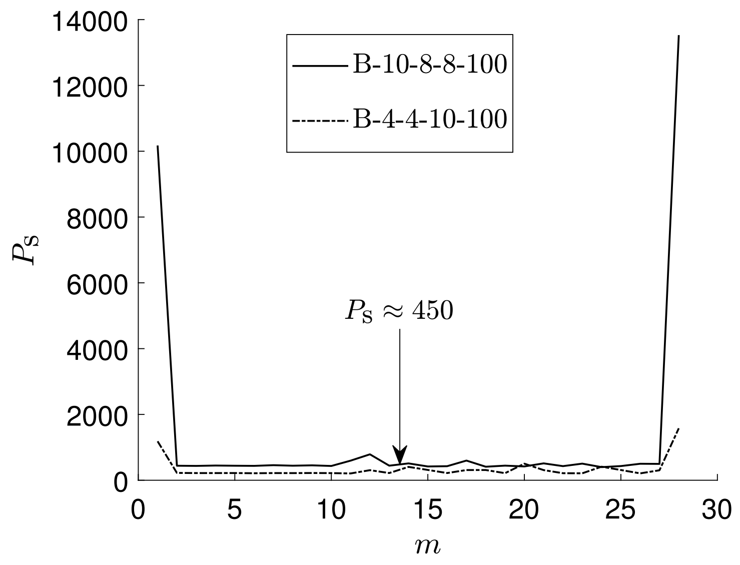


Figure 5.7: Distribution of number of suboptimal part-columns for the subproblems. m denotes the subproblem, and P_s denotes the amount of found suboptimal part-columns for the corresponding subproblem.

created for this problem is to some degree full of symmetries, making a large number of feasible solutions equally good. If constraints or costs which brake these symmetries are introduced, there is a big chance that the column generation will find better and even approximately optimal solutions.

5.2.4 Computational Performance

A motivation for applying column generation on the task mapping problem was the computational complexity of ILP and branch-and-bound. In Table 5.4 the running times for the problem instances are presented. Firstly, it should be mentioned that measuring and comparing wall clock time is debatable. This is due to the fact that there can be many simultaneous processes in the computer, making it difficult to know which program's running time is being measured. This being said, the wall time can give some insight into a programs grade of parallelism, through a comparison with the CPU time.

It is evident that the motivation was justified, as Gurobi's branch-and-bound solver clearly needed the most CPU time. However, because of its efficient parallelization, the elapsed real time was magnitudes lower than the CPU time. But ideally a mapping should not require almost nine hours. For grids with hundreds or thousands of cores, alternative methods to branch-and-bound must be explored. For the branch-and-bound solver it was clearly better to restrict the grid, which was expected. As seen in Table 5.3, changing the grid in this way did not increase the objective value significantly (from 82 to 85), giving motivation to this approach.

Instance	Wall time (hours)	CPU time (hours)
B-10-8-8-1	3.50	4.28
B-10-8-8-100	19.73	23.11
NB-10-8-8-1	0.03	0.04
NB-10-8-8-100	0.33	3.48
G-10-8-8	8.55	239.12
B-4-4-10-1	0.41	1.06
B-4-4-10-100	1.29	2.82
NB-4-4-10-1	0.05	0.55
NB-4-4-10-100	0.17	1.94
G-4-4-10	0.38	10.40

Table 5.4: Total running times. Wall time refers to the elapsed real time. CPU time refers to the time spent, by the CPU, processing instructions associated with the program.

Although the formulation with a block structure could improve the initial solution, it was the implementation of column generation which required the most computing time. Also, having suboptimal columns in the RMP increased the running time further. The increased number of variables in the RMP, as compared to instances where suboptimal columns were not saved, is likely the cause of this. Therefore there is a trade-off between running time and quality of the solution for implementations with a block structure. The implementation did not have a particularly high grade of parallelization, which is revealed by comparing the wall and CPU times. The reason for this is likely because of the simplicity of the individual subproblems. Not that many nodes needed exploration in the branch-and-bound trees and therefore a parallelization was unnecessary. As mentioned in Section 5.1, the individual subproblems were not solved in parallel. Tests where this is implemented would therefore be of interest.

The implementation without a block structure was significantly faster than all the others. The fast convergence and the reduced number of variables in the RMP explains this. It was also parallelized to a higher extent. This is because of the increased complexity of the subproblem. The comparison between having a block structure or not shows that having separable subproblems that are too easy to solve can become computationally problematic.

6 Conclusion

In this project an MBLP model for the deployment of processing functions onto a many-core grid, was created. The model was created by decomposing the processing graph into function sequences and using binary variables for describing the assignment of functions and associated connections, onto the grid. Successful mappings of processing functions was then done by solving the model to optimality using branch-and-bound. Knowing that the branch-and-bound method used would be computationally demanding as the problem size grows, the model created was then used as foundation for a Dantzig-Wolfe decomposition and column generation approach. Two decompositions were made, one with a block-diagonal structure in the constraint matrix of the subproblem, and one without this property.

The column generation method was then implemented in Julia using the modelling language JuMP and the solver Gurobi. The method was tested on both decompositions using a case graph with two different grids, and compared with optimal solutions found with the branch-and-bound method. In the column generation method, the procedure of saving intermediate suboptimal solutions from the subproblems, to improve convergence speed and resulting final solutions, was investigated. The results of the tests indicated that the column generation method was unsatisfactory in finding approximately optimal solutions to the current MBLP model.

For the case with a block-diagonal structure the calculated bounds for the RMP did not provide a good indication of the optimal objective value. It was also hard to conclude whether or not the algorithm was about to terminate using the bounds. Both restricting the grid and using suboptimal solutions from the subproblem, improved the convergence speed of the algorithm. By using a block structure it was possible to find improvements to the initial feasible solution for the larger grid. With suboptimal solutions used, further improvements could be made. In this case the improving part-columns were all chosen from the suboptimal solutions, motivating further investigation of this implementation. However, the resulting binary solution was far from optimal. The binary valued columns chosen in the solution to the IRMP showed that the subproblems associated with the input and output constraints could generate improvements, both with and without the use of suboptimal solutions. This led to the conclusion that introducing further constraints and differentiated costs in the model, to break symmetries in the problem, might improve the resulting binary solution produced by the column generation. For the instances with a block structure and the larger grid, the use of suboptimal solutions increased the computational running time significantly. But compared to the branch-and-bound method the column generation method is still faster. Reducing the grid size improved the running time. The low grade of parallelism for the implementation with a block structure suggests that a new implementation, where the subproblems are solved in parallel, should be tested.

Without a block-diagonal structure, the bounds gave roughly the same information, in terms of indication of the optimal objective value, as for the case with a block structure. However, without a block structure the bounds were more indicative as termination criteria for the column generation. Neither using suboptimal solutions nor restricting the grid improved the convergence rate for this case. But further testing on different grids should be done before drawing any conclusions regarding this. No improving solution was found for any of the instances without block structure. However, in terms of running time the implementation was superior to the others. It would therefore be interesting to further test this method with additional constraints or different costs in the model.

6.1 Further Work

It was shown that the column generation method can be used to find improvements to an initial feasible solution. The resulting binary solution was, however, far from optimal. The reason might be the vast amount of symmetries in the model created. As mentioned above, the current MBLP model could therefore be modified with respect to constraints and/or costs to remove some of these symmetries and increase the chances of finding improving columns.

A reformulation with a block structure was seen to improve the initial solution, but the implementation did not take advantage of a high grade of parallelism. The implementation can therefore be improved in terms of solving the subproblems in parallel, especially when the subproblems are easy enough to solve.

The implementation without a block-diagonal structure in the subproblem was very efficient in terms of computational running time. This implementation could therefore be used as a step in the branch-and-price framework [4, Section 1.3]. In this framework, when the column generation is finished the non-binary valued variables in the columns of the final RMP are branched upon and new reformulations are made in every node in the branch-and-bound tree.

Bibliography

[1] Adapteva. (2016). Epiphany-V: A 1024-core 64-bit RISC processor, [Online]. Available: `http://www.adapteva.com/announcements/epiphany-v-a-1024-core-64-bit-risc-processor/` (visited on 05/30/2019).

[2] K. Thulasiraman and M. N. S. Swamy, “Directed graphs”, in *Graphs: Theory and Algorithms*, Toronto, Canada: John Wiley & Sons, 1992, ch. 5, pp. 97–124.

[3] H. Orsila, E. Salminen, and T. Hämäläinen, “Recommendations for using simulated annealing in task mapping”, *Design Automation for Embedded Systems*, vol. 17, no. 1, pp. 53–85, 2013. [Online]. Available: `https://doi-org.proxy.lib.chalmers.se/10.1007/s10617-013-9119-0`.

[4] G. Desaulniers, J. Desrosiers, and M. Solomon, “A primer in column generation”, in *Column Generation*, G. Desaulniers, J. Desrosiers, and M. Solomon, Eds., New York, NY, USA: Springer, 2005, ch. 1, pp. 1–33.

[5] N. Andréasson *et al.*, *An Introduction to Continuous Optimization 3rd ed.* Lund, Sweden: Studentlitteratur, 2016.

[6] G. Sierksma, *Linear and Integer Programming: Theory and Practice 2nd ed.* Boca Raton, FL, USA: CRC Press, 2001.

[7] A. Schrijver, *Theory of Linear and Integer Programming.* Chichester, England: John Wiley & Sons, 1999.

[8] M. Lübbecke and J. Desrosiers, “Selected topics in column generation”, *Operations Research*, vol. 53, no. 6, pp. 1007–1023, 2005. DOI: `10.1287/opre.1050.0234`.

[9] R. Anbil, J. J. Forrest, and W. R. Pulleyblank, “Column generation and the airline crew pairing problem”, in *Proc. Internat. Congress of Mathematicians Berlin*, vol. 3, 1998, pp. 677–686. [Online]. Available: `https://elibm.org/article/10011622` (visited on 06/03/2019).

[10] JuliaOpt. (2019). JuliaOpt: Optimization packages for the Julia language, [Online]. Available: `http://www.juliaopt.org/` (visited on 05/25/2019).

[11] I. Dunning, J. Huchette, and M. Lubin, “JuMP: A modeling language for mathematical optimization”, *SIAM Review*, vol. 59, no. 2, pp. 295–320, 2017. DOI: `10.1137/15M1020575`.

[12] Gurobi Optimization, LLC. (2019). Gurobi Optimizer Reference Manual, [Online]. Available: `http://www.gurobi.com` (visited on 05/25/2019).

[13] JuliaOpt. (2019). Gurobi.jl, [Online]. Available: `https://github.com/JuliaOpt/Gurobi.jl` (visited on 05/25/2019).

[14] The Julia Project. (2019). The Julia Programming Language, [Online]. Available: `https://julialang.org/` (visited on 05/25/2019).

[15] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing”, *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: `https://doi.org/10.1137/141000671`.

[16] Intel. (2019). Intel® Xeon® Processor E5-2683 v3, [Online]. Available: `https://ark.intel.com/content/www/us/en/ark/products/81055/intel-xeon-processor-e5-2683-v3-35m-cache-2-00-ghz.html` (visited on 05/27/2019).