Simplex, Kolomgeneratie en Branch-and-Bound algoritmes voor Lineaire Problemen

Alexander Colman

Een thesis opgesteld voor de titel
Bachelor in de Fysica

Onder begeleiding van prof. dr. Bart Partoens

University of Antwerp
Faculty of Science

Departement Fysica
Universiteit Antwerpen
11 mei, 2025

Acknowledgement

I would like to thank my promoter prof. Bart Partoens for his great help in not only understanding the subjects, but also in coding the algorithms described in this thesis. Although it took some time to comprehend the matter, his guidance was exceptional and his patience boundless.

Contents

Abstract

This thesis explores optimization techniques, focusing on Linear Programming (LP). LP problems, characterized by linear objectives and constraints, are commonly solved using the Simplex method, which iteratively searches for an optimal solution along the edges of a convex polytope. For large-scale problems, column generation offers a scalable solution by iteratively solving a reduced master problem and a subproblem that introduces new variables based on reduced costs. However, such solutions are often non-integer, which limits their applicability in many real-world cases. To address this, the Branch-and-Bound method is used to enforce integrality by systematically exploring decision trees through branching on paths, arcs, or nodes. This thesis implemented the Simplex method, column generation, and Branch-and-Bound in Python using Gurobi, successfully solving the example problem in [2]. A comparative study of branching tactics, such as enforcing and omitting high- and low-flow arcs, and making whole-path constraints, showed varying effects on computation time. The results suggest that no single branching approach is universally optimal for the computation time; when enforcing the arcs in the path with the highest flow, the whole-path constraint is fastest, while the two other tactics are the fastest in all three other constraint types.
Additional works could build on this thesis' code and primitive findings. Especially researching the structure and length of a branching tree would be interesting.

In deze thesis worden optimalisatietechnieken onderzocht, waarbij de nadruk ligt op lineair programmeren (LP). LP-problemen, gekenmerkt door lineaire doelfuncties en beperkingen, worden gewoonlijk opgelost met behulp van de Simplex-methode, die iteratief zoekt naar een optimale oplossing langs de randen van een convexe polytoop. Voor grootschalige problemen biedt kolomgeneratie een schaalbare oplossing door het iteratief oplossen van een gereduceerd hoofdprobleem en een subprobleem dat nieuwe variabelen introduceert op basis van gereduceerde kosten. Dergelijke oplossingen hebben echter vaak een niet-gehele waarde, wat hun toepasbaarheid in veel praktijkgevallen beperkt. Om dit aan te pakken, wordt de Branch-and-Bound-methode gebruikt om oplossingen met gehele waarde af te dwingen door systematisch beslissingsbomen te verkennen via vertakkingen op paden, bogen of knooppunten. Deze thesis implementeerde de Simplex-methode, kolomgeneratie en Branch-and-Bound in Python met behulp van Gurobi, waarbij het voorbeeldprobleem in [2] met succes werd opgelost. Een vergelijkende studie van vertakkingstactieken, zoals het afdwingen en weglaten van paden met hoge en lage stromen, en het maken van hele-padbeperkingen, toonde verschillende effecten op de rekentijd. De resultaten suggereren dat geen enkele vertakkingsbenadering universeel optimaal is voor de rekentijd; bij het afdwingen van de bogen in het pad met de hoogste stroom is de gehele-padbeperking het snelst, terwijl de twee andere tactieken het snelst zijn bij alle drie de andere soorten beperkingen.
Bijkomende werken zouden kunnen voortbouwen op de code en primitieve bevindingen van deze Bachelorproef. Vooral het onderzoeken van de structuur en lengte van een vertakkingsboom zou interessant zijn.

1 Introduction

Optimization problems are central to decision making in various domains, from logistics and finance to manufacturing and network design. Linear Programming (LP) is one of the most widely used optimization techniques, offering efficient methods to solve problems that involve a linear objective function and linear constraints.

At its core, an LP problem consists of an objective function, either to be maximized or minimized, subject to a set of constraints that define a feasible region of solutions. Geometrically, this feasible region forms a convex polytope in an n-dimensional space, where optimal solutions are found at the "corner points" (intersections of the half-planes made by the constraints, see Section 2.1) of this polytope.

The Simplex method is one of the most commonly used algorithms for solving LP problems. It iteratively moves along the edges of the feasible region from one intersection to another, improving the objective function at each step, until the optimal solution is reached. The method efficiently handles constraints and variables by employing matrix formulations and pivot operations. [7]

When dealing with very large LP problems, explicitly formulating such an LP problem in a standard LP framework would require enumerating all possible paths, leading to an impractically large number of variables. Column generation provides an alternative approach by dynamically introducing only the most relevant variables into the model. Column generation iteratively solves a Restricted Master Problem, containing only a subset of variables (potential paths) and a subproblem that identifies new columns with reduced costs. [1]

Lastly, a way of making integer solutions is implemented. The solutions of column generation are not always of integer value: often, the optimal solution is the combined fraction of multiple paths. However, many real-world applications require integer solutions; Branch-and-Bound is a way to get integer solutions by adding constraints to the LP.

The goals of this work are to implement the Simplex algorithm, as well as the column generation algorithm via programming in Python. Specifically, the example provided in [2] will be fully implemented. Additionally, a small-scale examination of computation times will be performed using different (Branch-and-Bound) constraining tactics. Lastly, a randomized graph generation is also implemented, which can run column generation and add extra constraints. Because this thesis was made in a relatively short amount of time, a fully automated Branch-and-Bound algorithm wasn't made.

2 The Simplex method

The Simplex method is an algorithm to solve Linear Programs (LPs). To describe the algorithm, a concise introduction to Linear Programs is necessary.

2.1 Linear Programs

A Linear Program is an optimization problem for linear relations between variables. The LP can be written as a linear objective function that is to be minimized or maximized.

$$\begin{array}{ll} \text{Minimize/maximize} & \mathbf{c}^T\mathbf{x} \\ \text{Subject to:} & \mathbf{a_i}^T\mathbf{x} \geq b_i \quad i \in M_1 \\ & \mathbf{a_i}^T\mathbf{x} \leq b_i \quad i \in M_2 \\ & \mathbf{a_i}^T\mathbf{x} = b_i \quad i \in M_3 \\ & x_j \geq 0 \quad j \in N_1 \\ & x_j \leq 0 \quad j \in N_2 \end{array}$$

M_1, M_2 and M_3 are a set of finite indices, while b_i is a scalar and $\mathbf{a_i}$ a vector of n dimensions. The constraints are given by each row, for every value of i.
Note that either maximization or minimization of the objective function can be transformed into the other by the transformation $\mathbf{c}^T\mathbf{x} \rightarrow$ -$\mathbf{c}^T\mathbf{x}$. A constraint can also be transformed into another type of constraint with the following transformations:

$$\mathbf{a_i}^T\mathbf{x} \geq b_i \rightarrow -\mathbf{a_i}^T\mathbf{x} \leq -b_i$$
$$\mathbf{a_i}^T\mathbf{x} = b_i \rightarrow \mathbf{a_i}^T\mathbf{x} \geq b_i \text{ and } \mathbf{a_i}^T\mathbf{x} \leq b_i$$

Given this information, the canonical form of an LP can now be written as:

$$\begin{array}{ll} \text{Minimize/maximize} & \mathbf{c}^T\mathbf{x} \\ \text{Subject to:} & A\mathbf{x} \leq \mathbf{b} \\ \text{With:} & \mathbf{x} \geq 0 \end{array}$$

With A an $m \times n$ matrix, consisting of the coefficients of all m constraints and n decision variables.

These Linear Problems can be either feasible or infeasible. Their solution regions can be described geometrically: the feasible region is circumscribed by a convex polytope, with its dimensions related to the number of variables. In this context, "convex" means any two points inside the polytope can have a straight line between them that entirely lies within the polytope. For example, a feasible region for 2 variables and 3 constraints can be described by a convex polygon in a 2D space (a flat shape with straight edges). Every point outside this shape relates to the infeasible region. Figure 1 is a sketch of such an LP. In this case, the variables are x_1 and x_2; each linear constraint (of the form $a_1x_1 + a_2x_2 \leq b$) describes a half-plane. The feasible region is the intersection of all these half-planes, including the non-negativity constraints $x_1 \geq 0$ and $x_2 \geq 0$.

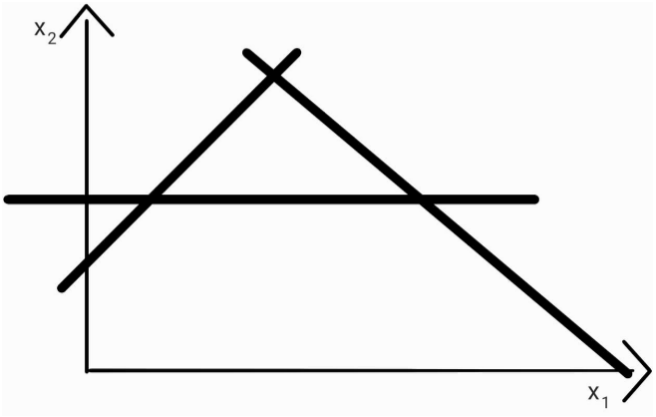

Figure 1: A sketch of an LP's solution space, with 2 variables and 3 constraints.

The polytope will at least have as many edges as the number of constraints on the variables. The optimal solution will always be found on one of the intersections of these edges: if an intersection is not the optimal solution, there will be an edge connecting this point to another intersection that improves the objective function. For a minimization problem, this means the objective function lowers; for a maximization problem, it increases. The optimal solution is dependent on the objective function.
Note that not all intersections between constraints will lead to a feasible solution. For instance, in Figure 1, the top intersection is in the *infeasible* region; as stated before, the feasible region is made from the cross-section of the half-planes of every constraint. Because the constraints are formulated as $a_1x_1 + a_2x_2 \leq b$, the "horizontal line" would cut off the region where the top intersection is. Figure 2 is the same as Figure 1, but with the feasible region highlighted and all intersection points between constraints marked.

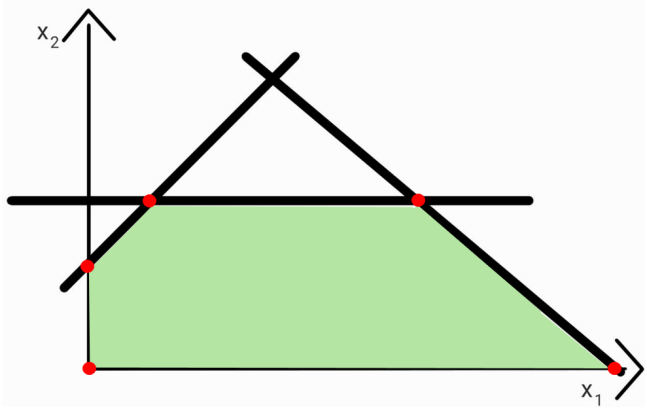

Figure 2: A sketch of the same LP as in Figure 1, but with the feasible region highlighted and all feasible intersections marked.

The previous statements can only be made because we're dealing with *Linear* Programs. In other words, LP problems have a solution space that has no

local minima. The edges that were discussed represent the paths between the extreme points of the solution, namely the intersections/corner points of the feasible region. Convexity of the polytope means the feasible region doesn't have "holes" or "indentations". From now on, an intersection/corner point in the solution space will be called a "vertex". [1] [2] [3]

2.2 The Simplex Algorithm

The Simplex method is an algorithm that finds the optimum or minimum of the LP by determining the correct vertex. Simplex searches along the edges of the feasible region for the optimal value, as was explained in Section 2.1.

The Simplex Algorithm consists of the following steps:

1. Start by determining a feasible vertex (often, the easiest one is the origin)
2. Next, the Simplex method searches for edges along which the objective function improves
3. When an ensuing vertex is found, look for another edge where the objective function improves. If none are found, this vertex is the optimal solution

Determining a starting feasible vertex is not always easy, especially when a large number of variables and constraints are inserted in the LP. Also, an LP can be "unbounded" and "too bounded": respectively, if a variable is unrestrained or if there is no possible solution, the Simplex Algorithm will break. In the case of an unbounded LP, the volume of the polyhedron becomes infinite, as the edges become infinitely long. [8]

2.2.1 An Example

To further explain the Simplex method, an example is needed. Imagine a baker has two recipes for baking cookies: chocolate chip cookies and peanut butter cookies. His goal is to maximize profits while ensuring he doesn't run out of ingredients:

$$\text{Max } \mathbf{c}^T\mathbf{x} = 5x_1 + 3x_2$$

The coefficients 5 and 3 represent the profit (arbitrarily chosen to be expressed in €) gained from respectively selling 1 unit of chocolate chip cookies (x_1) and peanut butter cookies (x_2). A constraint on this hypothetical scenario is:

$$2x_1 + x_2 \leq 100$$

In words, the baker needs 2 units of flour for each unit of chocolate chip cookies, as opposed to 1 unit of flour for each unit of peanut butter cookies. The total amount of flour present is 100 units. Another constraint the baker has to watch out for is the baking time. This, too, is a constraint on the objective function:

$$3x_1 + 2x_2 \leq 150$$

Each unit of chocolate chip cookies needs 3 time-units to bake. On the other hand, one unit of peanut butter cookies bakes in 2 time-units. The combined production of chocolate chip and peanut butter cookies must be limited so that

the total time required does not exceed the maximum available 150 time-units. Lastly, a non-negativity constraint needs to be added:

$$x_1, x_2 \geq 0$$

This constraint is to eliminate solutions where negative amounts of cookies have to be baked.
Now, the following LP is produced:

Max	$5x_1 + 3x_2$		(Maximizing profits)
Subject to:	$2x_1 + x_2$	≤ 100	(Units of flour)
	$3x_1 + 2x_2$	≤ 150	(The baker only has a limited time)
Non-negativity:	x_1, x_2	≥ 0	(Cannot bake negative #cookies)

Immediately, a problem arises: the constraints are expressed as inequalities, while the Simplex method requires a *standard form*-LP[1]. Standard form-LPs only have rows containing an equality sign, no inequality signs. To reshape this LP into standard form, the inequalities have to be made equalities *by introducing extra variables*. Each inequality "$\leq$" can be transformed as follows: $3x_1 + 5x_2 \leq 12 \rightarrow 3x_1 + 5x_2 + \alpha = 12$. Or, in the case of "$\geq$" inequalities: $4x_1 + x_2 \geq 9 \rightarrow 4x_1 + x_2 - \beta = 9$. Where α is often referred to as a "slack-variable" and β as a "surplus-variable", with $\alpha, \beta \geq 0$. These changes combine to the following LP:

$$\begin{array}{lll} \text{Max} & 5x_1 + 3x_2 & \\ & 2x_1 + x_2 & \leq 100 \\ & 3x_1 + 2x_2 & \leq 150 \\ & x_1, x_2 & \geq 0 \end{array} \rightarrow \begin{array}{lllll} \text{Max} & 5x_1 + 3x_2 & & & \\ & 2x_1 + x_2 & +x_3 & & = 100 \\ & 3x_1 + 2x_2 & & +x_4 & = 150 \\ & x_1, x_2, x_3, x_4 & & & \geq 0 \end{array} \quad (1)$$

Here, x_3 and x_4 are the slack variables. The LP is now in standard form and can be optimized by the Simplex method.

Solving the LP To determine the optimal number of each type of cookie to bake, the LP problem has to be solved. This is necessary to maximize profit while ensuring that resource constraints (in this case, time and ingredient availability) are satisfied.

The Simplex method follows a step-by-step procedure that examines the vertices of the feasible region to find the one that yields the highest objective value (profit). Firstly, the LP can be converted into a tableau form, which includes the objective function and all constraints. From this tableau, pivot operations can be performed to improve the solution (more on pivoting later). To prepare the current LP for the Simplex method, the objective function can be rewritten as follows[2]:

$$\mathbf{c}^T\mathbf{x} - 5x_1 - 3x_2 = 0$$

[1]The previous notation, canonical form, is a conventional way of writing LPs, but the standard form is necessary for Simplex.

[2]Although the interpretation of this reformulation is less intuitive, it is helpful in the tableau, as the value of the objective function can be displayed similarly to the values of the constraints. Its usefulness will become clear when solving the LP.

The basic variables are found by starting at a feasible vertex. The easiest one is the origin: $x_1 = 0$ and $x_2 = 0$. Now, $x_3 = 100$ and $x_4 = 150$ because of (1). So, x_3 and x_4 are the basic variables and x_1 and x_2 are the non-basic variables. The profit in this vertex is equal to 0, because $\mathbf{c}^T\mathbf{x} = 5x_1 + 3x_2$. This corresponds to baking no cookies, and no resources (flour and time) are used. Next, we will test if it is beneficial to bring a non-basic variable in the set of basic variables: will it increase the profit?

The $\mathbf{c}^T\mathbf{x}$-row contains the *reduced costs* of the non-basic variables: these convey whether increasing a variable (e.g. x_1 or x_2) would improve the objective. Negative values in this row (like -5 and -3) signal potential improvement in a maximization problem, indicating that the variable should enter the basis. Reduced costs will be further explored in Section 2.3.
To increase the objective function, all coefficients of the variables in the $\mathbf{c}^T\mathbf{x}$-row need to be ≥ 0 (a negative coefficient in the $\mathbf{c}^T\mathbf{x}$-row means a positive coefficient when rewriting the equation as $\mathbf{c}^T\mathbf{x} = \ldots$, meaning the profit could increase if more of units of the corresponding variable are added). This isn't the case right now, as both x_1 and x_2 have negative coefficients (-5 and -3 respectively). This means we need to **pivot** the variables (bring a variable into the basis, and take another out). To do this, the most negative variable in the $\mathbf{c}^T\mathbf{x}$-row (meaning the most positive variable in the objective function) gets chosen. This is $x_1 = -5$; the *entering variable*. The *leaving variable* gets chosen based on the Minimum Ratio Test (MRT):

$$\text{Ratio} = \frac{\text{Constraint value}}{\text{Coefficient of entering variable}} \tag{2}$$

This simple equation gives us that the ratio for the basic variables x_3 and x_4 respectively equals to $\frac{100}{2} = 50$ and $\frac{150}{3} = 50$ (using (3)). The slack-variables have an equal ratio, which means either can be chosen as the leaving variable. Here, x_3 will be chosen arbitrarily, because of the lower subscript[3].

Having defined the entering and leaving variables, the pivoting can begin. In terms of the basis-variables (one of which leaves the set), the LP now looks like this:

Basis	x_1	x_2	x_3	x_4	Value
x_3	2	1	1	0	100
x_4	3	2	0	1	150
$\mathbf{c}^T\mathbf{x}$	-5	-3	0	0	0

$\rightarrow$

Basis	x_1	x_2	x_3	x_4	Value
x_1	1	0.5	0.5	0	50
x_4	0	0.5	-1.5	1	0
$\mathbf{c}^T\mathbf{x}$	0	-0.5	2.5	0	250

(3)

In the tableau, the second and third row present the basis variables. These have a certain value, displayed in the last column. Columns 2-5 contain the coefficients of each variable per basis variable. The interpretations of these coefficients can be found in the paragraph **Interpreting the solution**.

Every element of the x_3-row gets divided by 2 during the pivoting; to swap

[3]Bland's rule says that when there are multiple possible x_i, the lowest subscript i should always be picked first (for both the entering and leaving variables). This helps omit endless cycling in the case of degeneracy; a basic variable has the coefficient 0, so a different basis can be achieved by pivoting, but the objective function is the same. Bland's rule thus aids in the case of multiple degenerate solutions all fitting the same objective function.

x_1 and x_3, every element needs to be divided by the coefficient of x_1. The following operations are then used to update the other 2 rows:

- New x_4-row = old row - (coefficient in old row for x_1)× (new pivot row)

$$\begin{aligned} x_{4_{\text{new}}} =& x_{4_{\text{old}}} - 3x_1 \\ =& (3, 2, 0, 1, 150) - 3 \times (1, 0.5, 0.5, 0, 50) \\ =& (0, -0.5, -1.5, 1, 0) \end{aligned}$$

- New $\mathbf{c}^T\mathbf{x}$-row = old row - (coefficient in old row for x_1)× (new pivot row)

$$\begin{aligned} (\mathbf{c}^T\mathbf{x})_{\text{new}} =& (\mathbf{c}^T\mathbf{x})_{\text{old}} - 5x_1 \\ =& (-5, -3, 0, 0, 0) - (-5) \times (1, 0.5, 0.5, 0, 50) \\ =& (0, -0.5, 2.5, 0, 250) \end{aligned}$$

In words, the new rows get calculated using their old row, minus the incoming variable row, which is multiplied by the coefficient of said variable in the old row.

x_2 is still a negative variable in the current LP, which means it has to be the next *entering variable*. The MRT (2) gives us for x_1 and x_4 respectively: $\frac{50}{0.5} = 100$ and $\frac{0}{0.5} = 0$. Thus, x_4 will be the next *leaving variable*. After dividing every element of the x_4-row by the coefficient of x_2, (3) is now:

Basis	x_1	x_2	x_3	x_4	Value
x_1	1	0.5	0.5	0	50
x_4	0	0.5	-1.5	1	0
$\mathbf{c}^T\mathbf{x}$	0	-0.5	2.5	0	250

$\rightarrow$

Basis	x_1	x_2	x_3	x_4	Value
x_1	1	0	2	-1	50
x_2	0	1	-3	2	0
$\mathbf{c}^T\mathbf{x}$	0	0	1	1	250

(4)

Where the other 2 rows are found via:

- New x_1-row = old row - (coefficient in old row for x_2)× (new pivot row)
- New $\mathbf{c}^T\mathbf{x}$-row = old row - (coefficient in old row for x_2)× (new pivot row)

$$\begin{aligned} \mathbf{c}^T\mathbf{x}_{\text{new}} =& \mathbf{c}^T\mathbf{x}_{\text{old}} - (-0.5)x_2 \\ =& (0, -0.5, 2.5, 0, 250) + 0.5 \times (0, 1, -3, 2, 0) \\ =& (0, 0, 1, 1, 250) \end{aligned}$$

A solution has been found where no coefficients are negative, meaning the solution is optimal! To maximize profits, the baker should keep to a ratio of 50 chocolate chip cookies to 0 peanut butter ones. In other words, the baker should only bake chocolate chip cookies. [9]

Final solution: $x_1 = 50$, $x_2 = 0$, $x_3 = 0$, $x_4 = 0$, $\mathbf{c}^T\mathbf{x} = 250$

Interpreting the solution The right-hand LP of (4) tells us that both x_1 and x_2 are in the basis of the solution, while x_3 and x_4 are not. Due to the theory explained in Section 2.1, variables not in the basis are always 0 at optimality. This is good, because the interpretation of the slack-variables x_3 and x_4 is that they measure unused resources in the original constraints. Optimally, they should be 0, so all available resources are fully used (meaning no leftover flour or baking time). The right-hand LP in (4) should be read like this:

- The coefficients of x_3 and x_4 in the basic variable rows portray the substitution rates between the basic variables and the non-basic variables: they depict how much the value of a basic variable would have to change if the non-basic variable were to increase by 1 unit. Geometrically, this would mean moving away slightly from the optimal solution. For example, the coefficient "2" in the x_1-row for variable x_3 means that if x_3 were to be increased by 1 unit, x_1 would have to decrease by 2 units[4] to maintain feasibility. The opposite would have to happen if x_4 were to increase by 1 unit: x_1 should be increased by 1 unit as well.
- The $\mathbf{c}^T\mathbf{x}$-row contains the *reduced cost* of the non-basic variables. The coefficients are both 1, meaning that if x_3 or x_4 were to increase by 1 unit, the objective function would decrease by 1. Since this is an optimization problem, this solution is optimal; x_3 and x_4 are both 0, meaning the objective function could only decrease by adding (one of) these variables into the basis. More on reduced costs in Section 2.3.

Figure 3 shows a plot of the feasible region, and all its vertices are marked. This plot makes it easy to see what exactly has been done in this Section. Firstly, the feasible vertex in the origin was chosen as a starting point. Then, after pivoting, the optimal solution was found in $x_1 = 50, x_2 = 0$. Because this model is small and simple, it is easy to check if this solution is correct; the third vertex is in the point $x_1 = 0, x_2 = 75$, where the objective function has a value of $\mathbf{c}^T\mathbf{x} = 5 \cdot 0 + 3 \cdot 75 = 225$. This is less than in the vertex found using Simplex, which means the solution is certainly correct! To conclude this example, the baker should make 50 chocolate chip and 0 peanut butter cookies to maximize profits, while using all available flour and baking time. If the baker does this, he would get a revenue of €250. [1] [3]

2.3 Matrix notation

Linear algebra enables us to write long and cumbersome LPs in a compact way. A canonical-form LP can be written as:

$$\text{Max } \mathbf{c}^T\mathbf{x} = \mathbf{c}_B^T\mathbf{x}_B + \mathbf{c}_N^T\mathbf{x}_N$$

Where B stands for "Basic" and N stands for "Non-basic". Essentially, the objective function is a sum of the objective function of basic variables and non-basic variables. This is subject to the constraints:

$$A\mathbf{x} = \mathbf{b} \Rightarrow \left[\ B \mid N\ \right] \cdot \begin{bmatrix} x_B \\ x_N \end{bmatrix} = \left[\ \mathbf{b}\ \right]$$

[4] An interpretation of this: if the unused flour were to increase by 1 unit, the amount of baked chocolate chip cookies should have to decrease by 2.

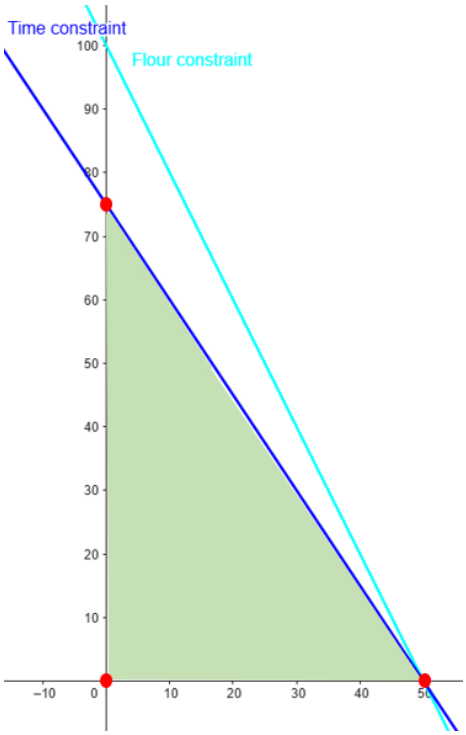

Figure 3: A plot of the feasible region, with the vertices marked. This plot was made in Geogebra [6].

In words, the constraint-coefficient matrix A can be divided into a basic and non-basic section. A is of $m \times n$ dimensions, with n the number of variables and m the number of constraints. B is the basis matrix: it consists of the columns of A that correspond to the basic variables (x_B). In Simplex, the number of basic variables is always equal to the number of constraints m. Therefore, B is a square matrix with dimensions $m \times m$. The basic variables can be derived by the following:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ \Rightarrow B\mathbf{x}_B + N\mathbf{x}_N &= \mathbf{b} \\ \Rightarrow B^{-1}(B\mathbf{x}_B + N\mathbf{x}_N) &= B^{-1}\mathbf{b} \\ x_B + B^{-1}N\mathbf{x}_N &= B^{-1}\mathbf{b} \\ x_B &= B^{-1}\mathbf{b} - B^{-1}N\mathbf{x}_N \end{aligned} \tag{5}$$

The objective function now becomes:

$$\mathbf{c}^T\mathbf{x} = \mathbf{c}_B^T\mathbf{x}_B + \mathbf{c}_N^T\mathbf{x}_N = \mathbf{c}_B^T(B^{-1}\mathbf{b} - B^{-1}N\mathbf{x_N}) + \mathbf{c}_N^T\mathbf{x}_N$$

$$\Rightarrow \mathbf{c}^T\mathbf{x} = \mathbf{c}_B^T B^{-1}\mathbf{b} + (-\mathbf{c}_B^T B^{-1} N + \mathbf{c}_N^T)\mathbf{x}_N \tag{6}$$

At last, two important equations have been found, each containing some aspects of the LP problem. Equation (5) portrays the *basic variables* $\mathbf{x_B}$, with $B^{-1}\mathbf{b}$ the values of the basic variables if all non-basic variables (x_N) are equal to 0. The basic variables' values change if x_N are > 0: each column in $B^{-1}N$ represents the substitution rates, which portray how much each x_B will decrease for a unit increase in the corresponding x_N (this is crucial for the Minimum Ratio Test (2)).
Equation (6) also contains two main elements. The first, $c_B^T B^{-1}\mathbf{b}$, calculates the value of the objective function at the current basic feasible solution (with $x_N = 0$). This part of (6) is called the *shadow price*. It provides information about the sensitivity of the objective function to a constraint's value (more on this in Section 3). The second part of the equation contains the *reduced cost* $c_N^T - c_B^T B^{-1} N$. The reduced cost "pulls" the LP towards the optimal solution: it tells us which x_N variable to bring into the set of basic variables. This happens when $c_N^T - c_B^T B^{-1} N < 0$ for minimization problems or $c_N^T - c_B^T B^{-1} N > 0$ for maximization problems. In other words, the reduced cost adapts the objective function in such a way that the optimal solution has the "best" value for the corresponding objective. If the reduced cost is non-positive for maximizations or non-negative for minimization problems, an optimal solution is found: the cost cannot increase or decrease any further. [1] [11] [3]

3 Duality of Problems

The concept of duality states that every LP problem (called the **primal problem**) has an associated **dual problem**. The dual problem is derived from the primal by the following principles:

- Every **constraint** in the primal corresponds to a **variable** in the dual problem
- Each **variable** in the primal correlates with a **constraint** in the dual problem
- **The constraint values** in the primal become **coefficients in the dual objective function**
- If the primal is a **maximization problem**, the dual is a **minimization problem** (and vice versa)

An example of how the dual problem corresponds to the primal problem, using the example demonstrated in Section 2.2.1:

$$\begin{array}{lcll} \text{Max} & 5x_1 + 3x_2 & & \\ \text{Subject to:} & 2x_1 + x_2 & \leq 100 & \\ & 3x_1 + 2x_2 & \leq 150 & \\ \text{Non-negativity:} & x_1, x_2 & \geq 0 & \end{array} \quad \rightarrow \quad \begin{array}{lcll} \text{Min} & 100y_1 + 150y_2 & & \\ & 2y_1 + 3y_2 & \geq 5 & \\ & y_1 + 2y_2 & \geq 3 & \\ & y_1, y_2 & \geq 0 & \end{array} \tag{7}$$

In this new formulation, the y_i represent the dual variables. These variables correspond to the constraints in the primal problem. The variables in the primal

have become constraints in the dual problem, while the coefficients of the dual constraints come from the columns of the primal constraint matrix:

$$\begin{matrix} x_1 & x_2 \\ \downarrow & \downarrow \\ \end{matrix} \begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix} \Rightarrow \begin{matrix} y_1 & \rightarrow \\ y_2 & \rightarrow \end{matrix} \begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}$$

The dual problem models the **cost of the resources**. Solving it gives insights into shadow prices; changing the value of a constraint by 1 will result in a change in the objective function equal to the shadow price of that constraint. To compactly state why and how the duality theory is important when solving LP problems:

- The **strong duality theorem** states that if an LP has an optimal solution, the corresponding optimal values of the primal solution are the same as those of the dual solution. This characteristic can be used to verify correctness.
- The optimal dual values indicate how much the primal objective function would improve if a constraint were relaxed (i.e. an increase in available resources). These are the **shadow prices**, as mentioned before.
- In large-scale LP problems, solving the dual problem often requires **less computational power** than solving the primal problem, especially when the amount of constraints is less than the number of variables.

The optimal value of the dual variables in (7) can be retrieved using the solution found in Section 2.2.1 (particularly: the right-hand LP of (4)). The optimal value of the dual variable corresponding to the i-th primal constraint is the respective coefficient in the final $\mathbf{c}^T\mathbf{x}$-row (the final solution). This final $\mathbf{c}^T\mathbf{x}$-row in the example contained: $[x_1, x_2, x_3, x_4] = [0, 0, 1, 1]$. This means $y_1 = 1$ (corresponding to the x_3-reduced cost[5]) and $y_2 = 2$ (corresponding to the x_4-reduced cost). These values mean that if the flour-constraint were to be increased by 1 unit ($100 \rightarrow 101$), the maximum profit would increase by €1. The same goes for the time-constraint. [1] [7] [3]

4 Column Generation

4.1 Setting/Motivation

When someone wants to optimize large-scale linear problems, the number of variables can become unreasonably large. Explicitly solving such enormous problems with the Simplex method would be cumbersome, impractical and time-consuming. Column generation is a helpful technique that only targets a small subset of variables; only new variables that could improve the current solution get generated.

[5]Note: in this particular example, the slack-variables are also non-basic variables. This means the value of x_3 and x_4 in the final $\mathbf{c}^T\mathbf{x}$-row have a dual interpretation; they are both the reduced cost of x_3 and x_4, and they're also the shadow price of the dual variables y_1 and y_2. The reduced costs apply to non-basic variables, and the shadow prices apply to constraints.

Many large-scale optimization problems can be modelled using graphs composed of nodes connected by arcs (or edges); Figure 4 shows an example. This thesis focuses on pathfinding problems within such graph structures, specifically on directed acyclic graphs (DAGs)[6]. The objective is typically to find an optimal path, or an optimal combination of paths, between designated source and sink nodes. A major distinction in the column generation formulation applied here, compared to some other network flow models, lies in the definition of variables. While variables in other models often represent flow on *individual arcs*, each variable in this column generation approach represents an *entire path* from source to sink. Although the number of constraints applied to the solution (such as a total time limit across chosen paths) might be relatively few and manageable, the number of potential paths can become enormous, even in moderately sized graphs. Consequently, explicitly formulating an LP that includes every possible path as a variable is often computationally infeasible. Column generation addresses this challenge by working with only a small subset of paths initially and iteratively generating new, potentially improving paths via a subproblem (as detailed in Section 4.4).

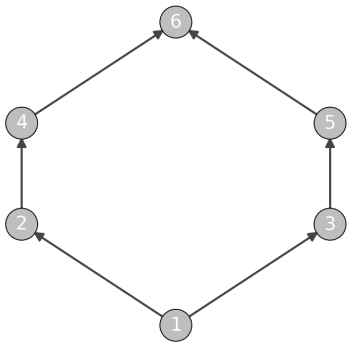

Figure 4: A simple example of a graph. This one has 6 nodes and 6 arcs. For pathfinding using this thesis' column generation approach, each arc gets a cost and time-cost.

In Simplex, a vertex corresponds to a particular basic solution (a set of active variables). In column generation, the set of selected paths corresponds to a basis in the Restricted Master Problem (explored in Section 4.3). These paths function similarly to individual variables in the Simplex tableau, with each path defining a column in the constraint matrix. [10] [3]

4.2 Reformulation of the LP: the Master Problem

We can reformulate an LP to a simpler subregion, called the *Master problem* (MP). Consider an LP of the form:

$$\begin{array}{lll} \text{Min} & \mathbf{c}^T\mathbf{x} & \\ \text{Subject to:} & \mathbf{A}\mathbf{x} & \geq \mathbf{b} \\ & \mathbf{x} & \in X \end{array} \tag{8}$$

where X defines a simpler feasible region within the original problem. In this thesis, X is assumed to be bounded, so its feasible region can be represented as a convex combination of a finite set of vertices, denoted as $\{\mathbf{x}_p\}_{p\in\mathcal{P}}$. Each $\mathbf{x}_p$ corresponds to a feasible path (with index p) from source to sink. $\mathcal{P}$ is the set of all feasible paths. $\mathbf{x}_p$ is a vector representing the values that the original problem variables ($\mathbf{x}$) take when they correspond to the path p. Any feasible

[6] In a directed graph, arcs have a defined direction from a starting node to an ending node. Acyclic means the graph contains no cycles, ensuring that no node can be visited twice within the same path.

solution $\mathbf{x} \in X$ can then be written as a weighted sum of its paths $\mathbf{x}_p$:

$$\mathbf{x} = \sum_{p \in \mathcal{P}} \lambda_p \mathbf{x}_p$$

where each λ_p is a coefficient, a weight, associated with a path x_p, with the convexity constraint being:

$$\begin{aligned} \sum_{p \in \mathcal{P}} \lambda_p &= 1 \\ \lambda_p &\geq 0 \\ \forall p &\in \mathcal{P} \end{aligned}$$

Substituting this representation of $\mathbf{x}$ into the original LP (8) yields the **Master Problem** (MP):

$$\begin{aligned} \text{Min} \sum_{p \in \mathcal{P}} \mathbf{c}^T \mathbf{x}_p \lambda_p \\ \text{Subject to:} \sum_{p \in \mathcal{P}} \mathbf{A}\mathbf{x}_p \lambda_p &\geq \mathbf{b} \\ \sum_{p \in \mathcal{P}} \lambda_p &= 1 \\ \lambda_p \geq 0, \quad \forall p &\in \mathcal{P} \end{aligned}$$

The challenge is that the number of vertices (feasible paths) $|\mathcal{P}|$ can still be enormous, making the explicit formulation of the MP impractical. To combat this, a **Restricted Master Problem** is introduced. [2]

4.3 The Restricted Master Problem (RMP)

Instead of evaluating all possible vertices (columns) in the Master Problem, column generation starts by solving a Restricted Master Problem (RMP). This includes only a small subset of these columns, denoted by $\mathcal{P}' \subseteq \mathcal{P}$:

$$\begin{aligned} \text{Min} \quad & \sum_{p \in \mathcal{P}'} \mathbf{c}^T \mathbf{x}_p \lambda_p \\ \text{Subject to:} \quad & \sum_{p \in \mathcal{P}'} \mathbf{A}\mathbf{x}_p \lambda_p \geq \mathbf{b} \\ & \sum_{p \in \mathcal{P}'} \lambda_p = 1 \\ & \lambda_p \geq 0, \quad \forall p \in \mathcal{P}' \end{aligned}$$

The RMP is a standard LP problem and can be solved using the Simplex method. [2]

4.4 The Subproblem

The central aspect of column generation is to efficiently identify whether any column in the full Master Problem exists that could improve the current optimal

solution of the RMP. This is done by solving a subproblem.

The subproblem uses the dual variables (shadow prices) obtained from the optimal solution of the RMP. $\boldsymbol{\pi}$ is the vector containing the dual variables associated with the constraints $\sum_{p\in\mathcal{P}'} \mathbf{A}\mathbf{x}_p\lambda_p \geq \mathbf{b}$. The dual variable π_0 is associated with the convexity constraint $\sum_{p\in\mathcal{P}'} \lambda_p = 1$. Then, the reduced cost $\overline{c}_p$ of a variable λ_p in the Master Problem is given by:

$$\overline{c}_p = (\mathbf{c}^T\mathbf{x}_p) - \pi^T(\mathbf{A}\mathbf{x}_p) - \pi_0$$

As stated in Section 2.3, a column corresponding to λ_p can improve the RMP solution if its reduced cost for a minimization problem is negative ($\overline{c}_p < 0$). Thus, for a minimization problem, the subproblem aims to find a path $\mathbf{x}_p \in X$ that minimizes this reduced cost:

$$\begin{array}{ll} \text{Min} & (\mathbf{c}^T\mathbf{x}) - \pi^T(\mathbf{A}\mathbf{x}) - \pi_0 \\ \text{Subject to:} & \mathbf{x} \in X \end{array}$$

This can be rewritten as:

$$\begin{array}{ll} \text{Min} & (\mathbf{c}^T - \pi^T\mathbf{A})\mathbf{x} - \pi_0 \\ \text{Subject to:} & \mathbf{x} \in X \end{array}$$

Now, the objective function of the subproblem is to minimize the "modified cost" of a vertex $\mathbf{x} \in X$, where the modification is based on the dual variables from the RMP. π_0 is not affected by which $\mathbf{x}$ minimizes the expression, so the minimization is solely based on $(\mathbf{c}^T - \pi^T\mathbf{A})\mathbf{x}$.
The "best" reduced cost[7] $\overline{c}^*_p$ is in this case the minimum of $\overline{c}_p$. If $\overline{c}^*_p \geq 0$ (the optimal reduced cost is non-negative), there are no columns in the full Master Problem that can improve the current RMP. Thus, the current RMP solution is optimal for the full Master Problem. However, if the minimum reduced cost is negative ($\overline{c}^* < 0$), the vertex $\mathbf{x}^*$ that yields this minimum corresponds to a new column with a negative reduced cost. This new column is added to the RMP, and the process is repeated. [2] [1]

4.5 The Column Generation Algorithm

Column generation, as previously described, solves large-scale LP problems using an efficient iteration algorithm. The algorithm goes as follows[8]:

1. **Initialize** the algorithm by formulating an initial RMP with an artificial variable for each constraint, each with a relatively high cost. These artificial variables are added because they guarantee a feasible starting solution (if the model itself is feasible). The high cost is assigned to ensure the column generation algorithm will prioritize driving the artificial variables to 0 in the final solution (if possible). Keeping them would result in a very high (undesirable) cost. This way of ensuring an initial solution is found, is called the "big M approach".

[7] In this context, "best" means the *most negative* reduced cost found among all potential columns. It's "best" because it represents the column that offer the greatest potential rate of improvement per unit of column added. Conceptually, the reduced cost is analogous to the slope of the objective function; the "best" means the steepest downhill slope. The opposite is true for maximization problems.

[8] The algorithm is explained in the context of a minimization problem.

2. **Solve RMP:** the current RMP is solved using the Simplex method. It is important to take note of the optimal primal solution (the values for the current λ_p variables), the objective function value, and the optimal dual variables π_i associated with the RMP's constraints.

3. Next, the **subproblem** is solved using the dual variables. The path $\mathbf{x}^*$, which minimizes the reduced cost, is found.

4. Lastly, a check for **optimality** is necessary:

   - If $\overline{c}^* \geq 0$, the current solution is optimal for the full Master Problem. The algorithm stops.
   - If $\overline{c}^* < 0$, generate a new column corresponding to $\mathbf{x}^*$ and add it to the RMP. Return to Step 2.

This iterative process will converge to the optimal solution of the Master Problem in a finite number of steps, if the model is feasible[9]. [5]

4.5.1 An example

The column generation can be more clearly understood with a simple example. Imagine a small warehouse with locations represented as nodes. A forklift needs to pick a certain item, which is stored in various locations (intermediate nodes) and deliver them to a final packing station (sink node), starting from a loading dock (source node). Each possible route the forklift driver can take is a path in the network. The goal is to find a combination of routes to pick up the item while minimizing the total cost. The cost could be interpreted in this case as the fuel consumed during transit (which can differ based on, for example, the elevation of the floor).
The goal (minimizing the fuel-costs) is subject to a constraint on the total time spent on all routes. The time-costs between the nodes can be viewed as the time it takes to traverse from one node to another.

[9] An infeasible model can be achieved by having either too restricting constraint values or too much constraints (or both) in the model.

Arcs	Costs	Time-costs
$1 \to 2$	2	5
$1 \to 3$	3	4
$2 \to 4$	4	6
$3 \to 4$	3	5

Table 1: All costs and time-costs, per arc.

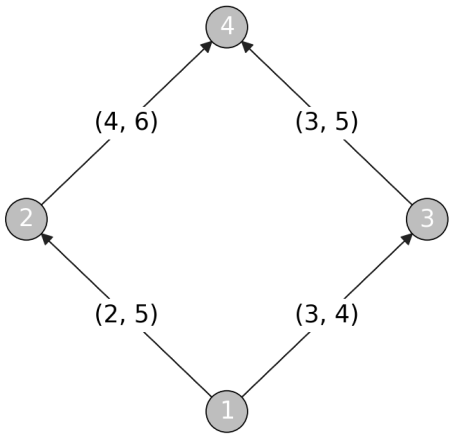

Figure 5: The model, plotted as a graph. The tuples (i, j) represent the costs and times for each arc: (cost, time). This graph was created using the "igraph" module in Python.

The challenge in this model is to find a path from node 1 to 4 while minimizing the costs and keeping to the time limit of 10. To solve this problem, each iteration will be expressed:

Iteration 0 To start, the Restricted Master Problem (RMP) gets added to the model, containing at least one column. To initialize this model, an artificial variable λ_{art} is introduced; this represents a "fake" path with a very high cost (100) that allows the RMP to be feasible[10]. The initial RMP is:

$$\begin{array}{lll} \text{Min} & 100 \cdot \lambda_{\text{art}} & \\ \text{Subject to:} & 0 \cdot \lambda_{\text{art}} \leq 10 & \text{(Time constraint)} \\ & \lambda_{\text{art}} = 1 & \text{(Convexity constraint)} \\ & \lambda_{\text{art}} \geq 0 & \end{array}$$

Solving this RMP is very simple:

$$\lambda_{\text{art}} = 1$$
$$\text{Objective value} = 100 \cdot 1 = 100$$

With π_0 the dual variable for the convexity constraint and π_1 the dual variable for the time constraint, they are found using:
$100 - \pi_1 \cdot 0 - \pi_0 \cdot 1 = 0 \Rightarrow \pi_0 = 100$. Because the time constraint is non-binding, π_1 is set to 0. For "$\leq$" constraints in minimization, the dual variables are non-positive, while the "$\geq$" constraints are non-negative. However, it should be noted that the standard MP form is to use "$\geq$" constraints. This means our time-constraint should actually be constructed as $-0 \cdot \lambda_{\text{art}} \geq -10$. In this case, it doesn't change the π_1, but in future iterations, it can.

[10]Here, this means the initial fake variable mitigates the time-constraint: its path-time gets set to 0, which means it always satisfies the constraint.

Iteration 1 To solve the subproblem, the previously found dual variables are used. In this part, the objective is to find a path $\mathbf{x} \in X$ which minimizes the reduced cost:

$$\text{Min} \quad (\mathbf{c}^T - \pi_1 \mathbf{t}^T)\mathbf{x} - \pi_0$$

Because $\pi_0 = 100$ and $\pi_1 = 0$ are known and constant, the subproblem is to simply find the "shortest" path (the one with the lowest cost) from node 1 to 4. Comparing the possible paths from 1 to 4 in the network:

Path 1: $1 \to 2 \to 4$:

- **Costs**: $c_{12} + c_{24} = 2 + 4 = 6$
- **Time**: $t_{12} + t_{24} = 5 + 6 = 11$
- **Reduced cost**: $6 - 0 \cdot 11 - 100 = -94$

Path 2: $1 \to 3 \to 4$:

- **Costs**: $c_{13} + c_{34} = 3 + 3 = 6$
- **Time**: $t_{13} + t_{34} = 4 + 5 = 9$
- **Reduced cost**: $6 - 0 \cdot 9 - 100 = -94$

Both paths have the same minimum reduced cost of -94. This means either can be added to the RMP as a new column. Just like in the Simplex method, the path with the lowest variable will be added (Path 1: $1 \to 2 \to 4$).

Because the minimum reduced cost is negative, the process needs to be repeated.

Iteration 2 The RMP now contains two variables: λ_{art} and λ_1 (path 1). The updated RMP is now:

$$\begin{array}{lrll} \text{Min} & 100 \cdot \lambda_{\text{art}} + 6 \cdot \lambda_1 & & \\ \text{Subject to:} & 0 \cdot \lambda_{\text{art}} + 11\lambda_1 & \leq 10 & \text{(Time constraint)} \\ & \lambda_{\text{art}} + \lambda_1 & = 1 & \text{(Convexity constraint)} \\ & \lambda_{\text{art}}, \lambda_1 & \geq 0 & \end{array}$$

The fake variable λ_{art} can be substituted by $\lambda_{\text{art}} = 1 - \lambda_1$ and the time constraint can be reformulated, so it uses the $\geq$ constraint:

$$\begin{array}{lrll} \text{Min} & 100 \cdot (1 - \lambda_1) + 6 \cdot \lambda_1 = & & \\ & 100 - 94 \cdot \lambda_1 & & \\ \text{Subject to:} & -11 \cdot \lambda_1 & \geq -10 & \text{(Time constraint)} \\ & 1 & = 1 & \text{(Convexity constraint)} \\ & \lambda_1 & \leq 1 & \\ & \lambda_1 & \geq 0 & \end{array}$$

Solving this new RMP:
To minimize $100 - 94 \cdot \lambda_1$, λ_1 has to be maximized. The constraint $11 \cdot \lambda_1 \leq 10$ gives $\lambda_1 \leq 10/11$. Seen as λ_1 is maximized if it has the value $10/11$, we also know that $\lambda_{\text{art}} = 1 - \lambda_1 = 1 - 10/11 = 1/11$. The objective value is thus $100 \cdot (1/11) + 6 \cdot (10/11) = 160/11 \approx 14.55$.

The dual variables of this solution are:
For λ_{art}: $100 - \pi_1 \cdot 0 - \pi_0 \cdot 1 = 0 \Rightarrow \pi_0 = 100$
For λ_1: $6 - \pi_1 \cdot 11 - \pi_0 \cdot 1 = 0 \Rightarrow 6 - 11\pi_1 - 100 = 0 \Rightarrow \pi_1 = -94/11$

Iteration 2: solving the subproblem The new, modified costs for each arc are calculated with $c'_{ij} = c_{ij} + (94/11) \cdot t_{ij}$.

$$\begin{aligned}
c'_{12} = \quad & c_{12} + (94/11) \cdot t_{12} \quad = 2 + (94/11) \cdot 5 \quad = 492/11 \quad \approx 44.73 \\
c'_{13} = \quad & c_{13} + (94/11) \cdot t_{13} \quad = 3 + (94/11) \cdot 4 \quad = 409/11 \quad \approx 37.18 \\
c'_{24} = \quad & c_{24} + (94/11) \cdot t_{24} \quad = 4 + (94/11) \cdot 6 \quad = 608/11 \quad \approx 55.27 \\
c'_{34} = \quad & c_{34} + (94/11) \cdot t_{34} \quad = 3 + (94/11) \cdot 5 \quad = 503/11 \quad \approx 45.73
\end{aligned}$$

To find the minimum reduced cost, the modified costs are used for their respective path:

Path 1: $1 \to 2 \to 4$:

- **Modified cost**: $c'_{12} + c'_{24} = 492/11 + 608/11 = 1100/11 = 100$
- **Reduced cost**: $100 - 100 = 0$

Path 2: $1 \to 3 \to 4$:

- **Modified cost**: $c'_{13} + c'_{34} = 409/11 + 503/11 = 912/11 \approx 89.91$
- **Reduced cost**: $912/11 - 100 = -188/11 \approx -17.09$

The minimum reduced cost is $-188/11$ (associated with path 2). Because this is the most negative, path 2 gets added as a new column to the RMP.

Iteration 3 The new RMP now consists of three variables:

$$\begin{aligned}
\text{Min} \quad & 100 \cdot \lambda_{\text{art}} + 6 \cdot \lambda_1 + 6 \cdot \lambda_2 \\
\text{Subject to:} \quad & 0 \cdot \lambda_{\text{art}} + 11\lambda_1 + 9 \cdot \lambda_2 \quad \leq 10 \quad \text{(Time constraint)} \\
& \lambda_{\text{art}} + \lambda_1 + \lambda_2 \quad = 1 \quad \text{(Convexity constraint)} \\
& \lambda_{\text{art}}, \lambda_1, \lambda_2 \quad \geq 0
\end{aligned}$$

Again, λ_{art} gets substituted: $\lambda_{\text{art}} = 1 - \lambda_1 - \lambda_2$.

$$\begin{aligned}
\text{Min} \quad & 100 \cdot (1 - \lambda_1 - \lambda_2) + 6 \cdot \lambda_1 + 6 \cdot \lambda_2 \\
& = 100 - 94 \cdot \lambda_1 - 94\lambda_2 \\
\text{Subject to:} \quad & 11 \cdot \lambda_1 + 9\lambda_2 \quad \leq 10 \quad \text{(Time constraint)} \\
& \lambda_1 + \lambda_2 \quad \leq 1 \quad \text{(Convexity constraint)} \\
& \lambda_1 + \lambda_2 \quad \geq 0
\end{aligned}$$

Solving this RMP is similar to the previous iteration. Now, $94 \cdot \lambda_1 + 94 \cdot \lambda_2$ needs to be maximized. The time constraint for λ_1 is more restrictive: $11 \cdot \lambda_1 \leq 10 \Rightarrow \lambda_1 \leq 10/11$, because for λ_2 it is: $9 \cdot \lambda_1 \leq 10 \Rightarrow \lambda_1 \leq 10/9$. This last inequality also shows how λ_{art} would be equal to $1 - 10/9 = -1/9$, making path 2 alone violate the convexity constraint. However, if the paths λ_1 and λ_2 were to be combined (and their weights altered), a minimal solution can be found.

Consider the boundary of the time constraint: $11 \cdot \lambda_1 + 9 \cdot \lambda_2 = 10$. In order to maximize the convexity constraint $\lambda_1 + \lambda_2 = 1$ ($\Rightarrow \lambda_1 = 1 - \lambda_2$):

$$\begin{aligned}
11 \cdot (1 - \lambda_2) + 9 \cdot \lambda_2 &= 10 \\
11 - 2 \cdot \lambda_2 &= 10 \\
\lambda_2 &= 0.5 \\
\Rightarrow \lambda_1 &= 0.5
\end{aligned}$$

So, $\lambda_1 = 0.5 = \lambda_2$ and $\lambda_{\text{art}} = 0$.
The objective value is $6 \cdot 0.5 + 6 \cdot 0.5 = 6$.
The dual variables for this solution are: For λ_1: $6 - \pi_1 \cdot 11 - \pi_0 \cdot 1 = 0$
For λ_2: $6 - \pi_1 \cdot 9 - \pi_0 \cdot 1 = 0$
Equating the two equations:

$$\begin{aligned} 6 - \pi_1 \cdot 11 - \pi_0 &= 6 - \pi_1 \cdot 9 - \pi_0 \\ \Rightarrow \pi_1 &= 0 \end{aligned}$$

Substituting π_1 into the first equation:

$$\begin{aligned} 6 - 0 \cdot 11 - \pi_0 &= 0 \\ \Rightarrow \pi_0 &= 6 \end{aligned}$$

Iteration 3: solving the subproblem The modified costs in this iteration are very simple, because $\pi_1 = 0$.

Path 1: $1 \to 2 \to 4$:

- **Modified cost**: $c''_{12} + c''_{24} = 2 + 4 = 6$
- **Reduced cost**: $6 - 6 = 0$

Path 2: $1 \to 3 \to 4$:

- **Modified cost**: $c''_{13} + c''_{34} = 3 + 3 = 6$
- **Reduced cost**: $6 - 6 = 0$

The minimum reduced cost is 0 for both paths, which means the optimal solution has been found!

To be certain this solution fits the criteria, a check is done if all constraints are satisfied:

- **Time**: $11 \cdot 0.5 + 9 \cdot 0.5 = 10 \leq 10$
- **Convexity**: $0.5 + 0.5 = 1$
- **Non-negativity**: $0.5, 0.5 \geq 0$

All constraints are satisfied! This means the optimal solution is to combine paths 1 and 2, each with a weight of 0.5. In the context of the warehouse, this could mean that each route is to be taken half of the time, or if the warehouse has multiple forklifts, this could imply that half of the overall flow needs to be assigned to each of these paths. [2]

5 Branch-and-Bound

Column generation solves the linear relaxation of the Master Problem, but it often leads to non-integer optimal solutions. Though, many real-world problems require integer solutions. For instance, the optimal solution of the model in Section 4.5.1 says both path 1 and path 2 have a flow of 0.5. One could argue that, in a warehouse, the route would be traversed multiple times, meaning this fraction gives an average utilization or expected cost. This means the forklift could, for example, alternate between both paths over the course of multiple iterations. However, if the LP was modelling employee scheduling, delivery truck

assignment, flight scheduling,... the non-integer solution wouldn't be as useful. The Branch-and-Bound process makes sure an integer solution is found (eventually) by applying a new constraint at fractional flows. This constraint can either be "$= 1$", or it can be "$= 0$". In words, this means the node or arc that's branched on is, respectively, either enforced to be in the solution or excluded from the solution. In essence, the flow of a node or arc can be put to 1, to ensure it is kept in the solution, or it can be put to 0, to omit it. Additionally, this branching is not limited to only one node or arc per constraint; however many nodes/arcs can be inserted in the same constraint. To do this, one could either say the sum of the flows through multiple arcs is "$= 0$" or "≥ 1". Note that in the case of multiple components, the enforcing constraint should be "more than or equal to 1": in the case of enforcing two arcs, their sum of flows could also equal "2" if both are in the final solution.

When adding k new constraints, it is advised to always add k fake variables to the model as well. This way, the chance of an infeasible model is smaller, as a first solution is guaranteed to be found (given the constraints themselves don't make the model infeasible). [12] [1]

5.1 The Branch-and-Bound Algorithm

The Branch-and-Bound algorithm is essentially nothing more than adding new constraints and re-running the column generation with these new constraints (and their respective fake variables). It is possible that after branching, the new solution has fractional flows too. In that case, new branches should be added accordingly.

An example of branching on a solution can be implemented on the example used in Section 4.5.1: one could, for example, compare branching on $(1 \to 2)$ and $(1 \to 3)$. The same algorithm can be followed, but this time another constraint is added: either[11] $(1 \to 2) = 1$ or $(1 \to 3) = 1$.
The Python code used to calculate the column generation (with branching) in Section 6.1 is used to solve this example[12]; the same steps as in Section 4.5.1 need to be performed, but with an extra constraint. The following solutions are then found:

$$(1 \to 2) = 1: \quad \lambda_0(\text{Value: } 0.9091) = (1 \to 2), (2 \to 4)$$
$$(1 \to 3) = 1: \quad \lambda_0(\text{Value: } 1) = (1 \to 3), (3 \to 4)$$

Branching on $(1 \to 3)$ results in a path that has value 1: the path $(1 \to 3), (3 \to 4)$ satisfies the time-constraint and has an objective value of 6.
Inserting $(1 \to 2)$ in the constraint, does not satisfy the time-constraint: $11 > 10$. This is why its solution has a non-integer flow.

Eventually, a Branch-and-Bound tree can be constructed, where each two leaves represent either a "$= 0$", or a "$\geq$ or $= 1$" decision for a specific constraint.
However, due to a limited amount of time, the Branch-and-Bound algorithm won't be explored fully. Further works could both implement an automated

[11] These constraint values and arcs are chosen arbitrarily.
[12] This code can be found in the Appendix.

code that implements this algorithm, as well as research multiple aspects of the algorithm:

- If the amount of constraints m is larger than 1, the order of which node(s)/arc(s) to branch on, is possibly not arbitrary; the computation time or iterations needed to solve a branched LP could differ based on this order.
- A tree-search mechanic can be created to make the Branch-and-Bound algorithm stop early, or to focus a specific branch in the tree based on the objective function's value.

6 Implementation of the theory

6.1 Computationally solving a model

This thesis was initially scoped to understand and implement the Simplex, column generation and the Branch-and-Bound algorithms. To fully grasp how these algorithms work, the first objective was to make a program that can solve the example found in [2] (pages 1-7). A schematic illustration of the model is shown in Figure 6.

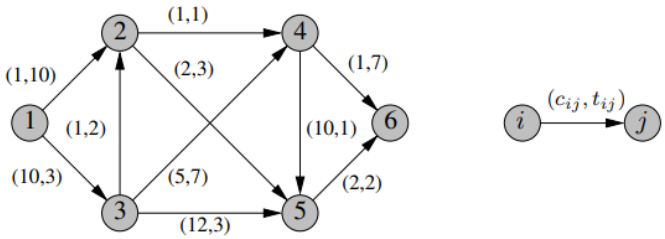

Figure 6: The time-constrained shortest path problem, as depicted in [2].

To solve this model computationally, Python was used, aided by importing the module "gurobipy". This module incorporates an optimize() method that uses Simplex to optimize an LP. The code, along with an extensive description) can be found in the Appendix.
The first iteration (named BB0) implements column generation on the shortest path problem shown in Figure 6. By adding a fake variable y_0 with a high cost (100) to the model, the convexity constraint is secured. In this case, the time constraint is not implemented through a secondary fake variable: by setting the path time of λ_0 to 0, the algorithm can continue because it always satisfies the time constraint. The initial solution found with the Simplex method is $y_0 = 1$, with an objective value (denoted as $\overline{z}$) of 100 and a minimum reduced cost of -97. From BB0.2 on, column generation iterates optimizations until the minimum reduced cost reaches 0.
The optimal solution is found for the combination $\lambda_{13256} = 0.2$ and $\lambda_{1256} = 0.8$. Note that the objective function reduces per iteration. Table 2 shows how the solution of the model evolves through each iteration of the column generation.

Iteration	Master solution	$\overline{z}$	π_0	π_1	$\overline{c}^*$
BB0.1	$y_0 = 1$	100.0	100.00	0.00	-97.0
BB0.2	$y_0 = 0.22$, $\lambda_{1246} = 0.78$	24.6	100.00	-5.39	-32.9
BB0.3	$\lambda_{1246} = 0.6$, $\lambda_{1356} = 0.4$	11.4	40.80	-2.10	-4.8
BB0.4	$\lambda_{1246} = \lambda_{13256} = 0.5$	9.0	30.00	-1.50	-2.5
BB0.5	$\lambda_{13256} = 0.2$, $\lambda_{1256} = 0.8$	7.0	35.00	-2.00	0

Table 2: Summary of iterations for BB0 iteration found in [2].

This solution is non-integer; to get the lowest possible cost while satisfying the time constraint, fractions of two paths need to be taken. A non-integer solution can be found by using the Branch-and-Bound algorithm: iterations BB1, BB2, BB3 and BB4 all include some form of branching constraints, with a purpose of making the solution from BB0 integer. Table 3 shows the details of these branching decisions.

- The **BB1** iteration sets a branching solution on two arcs at once: $x_{13} + x_{32} = 0$. In words, this constraint "removes" the arcs $1 \to 3$ and $3 \to 2$ from available list of arcs through constraints. However, the initial fake variable y_0 needs to have a higher cost, because y_0 is too interfering with the problem: the final solution still contains the fake variable.
  After setting the initial cost to 1000, the new solution has a cost of 14, which is higher than the previous cost found in BB0. This is because the constraint makes it so the "most optimal path combinations" isn't included[13] (because of the integer-requirement).
- **BB2** is the opposite of BB1; this time the arcs $1 \to 3$ and $3 \to 2$ are enforced to be in the solution (specifically, their sum of flows should be "≥ 1"). An objective value of 9 is found, which is lower than if $1 \to 3$ and $3 \to 2$ get left out of the solution. However, the values of the path-variables are still non-integer. BB3 and BB4 will investigate what enforcing and removing the arc $1 \to 2$ does.
- **BB3** adds two constraints: one constraint requires both $1 \to 3$ and $3 \to 2$ to be in the solution (just like in BB2), while the other constraint keeps $1 \to 2$ out of the solution. This branch results in the *lowest objective value* of all branching!
- **BB4** shows what the model does if incompatible branching constraints are added: here, both $(1 \to 2)$ and $(1 \to 3$ and $3 \to 2)$ are enforced. This results in an infeasibility-problem. Note that the solution still contains the fake variable (and the other path in the solution is the same as in BB1, where there was an infeasibility too).

[13]Note: this isn't always the case; in a different LP, there could be multiple paths with the same objective value (just like what was found in Section 5.1: two solutions both have an objective value of 6.

Iteration	Master Solution	$\overline{z}$	π_0	π_1	π_2	$\overline{c}^*$
BB1:	BB0 and $x_{13} + x_{32} = 0$					
BB1.1	$y_0 = 0.067$, $\lambda_{1256} = 0.933$	11.3	100	-6.33	-100	0
BB1.2	*Initial cost increased to 1000*					
	$y_0 = 0.067$, $\lambda_{1256} = 0.933$	71.3	1000	-55.4	-1000	-57.3
BB1.3	$\lambda_{12456} = 1$	14	1000	-70.43	-1000	0
BB2:	BB0 and $x_{13} + x_{32} \geq 1$					
BB2.1	$\lambda_{1246} = \lambda_{13256} = 0.5$	9	15	-0.67	3.33	0
BB3:	BB2 and $x_{12} = 0$					
BB3.1	$\lambda_{13256} = 1$	15	15	0	0	-2
BB3.2	$\lambda_{13246} = 1$	13	0	0	-1000	0
BB4:	BB2 and $x_{12} = 1$					
BB4.1	$y_0 = 0.067$, $\lambda_{1256} = 0.933$	111.3	100	-6.33	100	0
	Stopping due to infeasibility					

Table 3: Details of the Branch-and-Bound decisions.

6.2 Duration of different branching tactics

In this section, the calculation time of different branching tactics is explored. There are multiple ways to make branching constraints: one could branch on some arcs with a fractional flow separately, or put multiple arcs into one branching constraint. A third way to branch is to set a whole path with a fractional flow in the branching constraint: either the whole path is enforced, or it's excluded. To get a better understanding of the time needed to calculate a solution for different branching tactics, a large LP is necessary. The previous examples are too small to get a meaningful answer when trying different branching tactics, as the computing time needed is always less than 0.01 seconds.

An initial thought was to copy the graph-frame from a large workflow model. The dataset from a project called Montage was imported; this project uses a workflow management system called Pegasus and incorporates 1000 nodes and 2484 arcs in its graph. The Montage project is designed to create custom mosaics of astronomical images, and its workflow graph can be found in [4]. A plot of this dataset can be seen in Figure 7; here, every node represents a different task to complete the goal of creating the mosaics.
Although the graph only has one sink, it has 166 sources. This means an adjustment has to be made to the graph; because the model is only used to benchmark calculation times, it's acceptable to add a dummy node. This dummy node will be added manually, connecting to all original source nodes. This causes the model to only have one source node and one sink node. Additionally, a random cost and time-cost between 1 and 20 are given to every arc in the model.

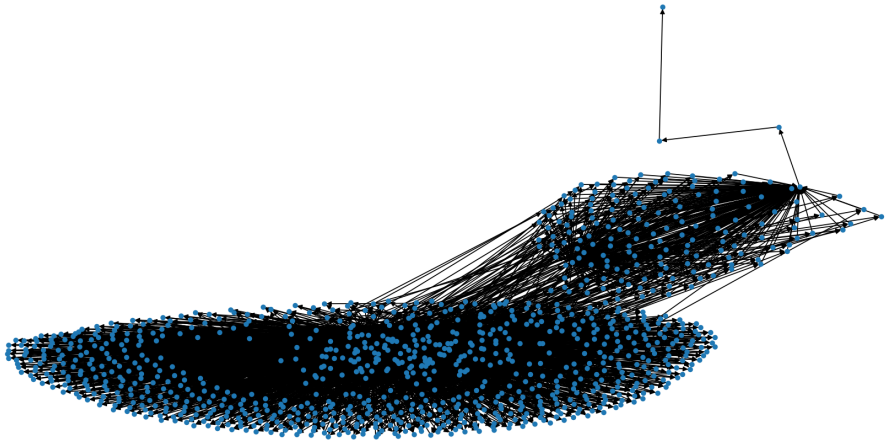

Figure 7: A messy plot of the example data [4]. Each dot represents a node (job/task).

While this dataset initially appeared promising, applying the column generation code used in Section 6.1 revealed that it was not worthwhile. The paths found by column generation were always approximately of length 6, which is about the same length as the ones found in Section 6.1. This graph was initially imported because of its large amount of nodes and arcs, but some short paths rendered the model not that useful.

To compare the computation times between the different branching tactics, a new graph is needed. Taking some inspiration from the example solved in Section 6.1, a model is created as seen in Figure 8, using the Python module networkx. Starting from the source node, each "step" towards the sink node has two possible arcs: every node is connected to the two succeeding nodes and given a random cost and time-cost between 1 and 20.

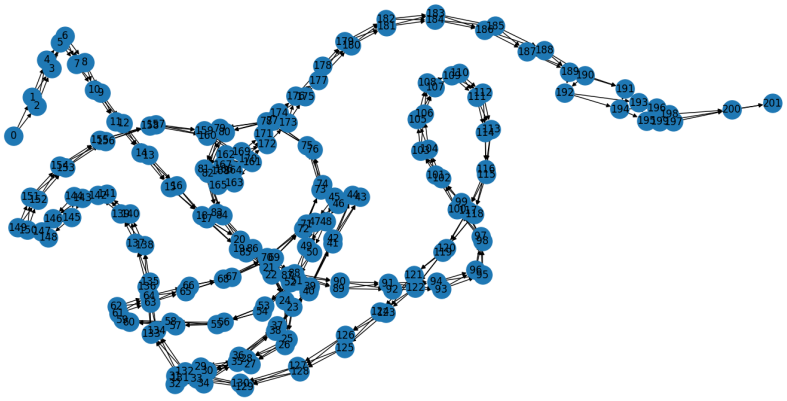

Figure 8: The newly created model, containing 202 nodes and 400 arcs.

This model is guaranteed to have a lot of possible paths, as every "step" has a choice of 2 nodes, depending on their cost and time-cost. At last, a large enough model can be used to distinguish the calculation times. Now, the model can be solved.
As in Section 6.1, a fake variable is created to secure the convexity constraint. Its path-time is set to 0, so the least-cost path is taken (the cost is to be minimized) and its cost is 10000. The time-limit is set to 1000 (this is an arbitrary number; any other would work. Although an integer solution may not always be found, depending on the time-costs and time-limit). An initial solution is calculated with Simplex.
Afterwards, column generation is run, using this initial solution. A fractional, optimal solution is found[14]: a combination of paths λ_5 and λ_6 (see next page), with an objective value of 626.24. 96 of the 100 arcs have a flow of 1; they are fully in the optimal solution. The other 4 arcs have a non-integer flow. Because this solution is fractional, Branch-and-Bound can be utilized. However, this section's goal is not to find a solution: the objective is to compare computation times of different branching tactics.

One arc per constraint Two arcs in the path λ_6 with a fractional flow get their own, separate branching constraint. These constraints can either be "$= 1$" or "$= 0$". The chosen arcs are $(63 \to 65)$ and $(65 \to 68)$[15], both with a constraint "$= 1$" ($x_{63\to65} = 1, x_{65\to68} = 1$); these constraint values mean both arcs are enforced.
The arcs were chosen based on arc flows: there are 8 arcs where a non-integer flow is found[16]. Not every non-integer arc is constrained: only half are selected to ensure that the third branching tactic, which branches on an entire path, remains meaningful. If all non-integer arcs were constrained individually, path-based branching would no longer offer a distinct strategy.
Running the column generation after adding these two constraints gives a solution containing two paths with non-integer flow. Under "normal" circumstances, another Branch-and-Bound would happen after finding a non-integer solution, but in this context, the solutions are not of importance; finding the time needed to calculate one round of branching is the main objective.

Two arcs in a constraint In this tactic, the two constraints from before are combined into a single one. When placing two arcs into one constraint, its value has to change from "$= 1$" to "≥ 1" ($x_{63\to65} + x_{65\to68} \geq 1$). This ensures that the sum of the flows through both arcs must be greater than or equal to 1 (at least one of the arcs has to be used). However, this also means the constraint is satisfied even if only one of the arcs is present in the path. To strictly enforce both arcs, the constraint is strengthened: the constraint is now "$x_{63\to65} + x_{65\to68} \geq 2$".

Whole path as a constraint A path can be fully omitted or enforced using constraints. Just like in the previous tactic, multiple arcs will be joined into one constraint. This time, the number of arcs is equal to the number of arcs in the path, and the constraint value is "≥ 100". This means the *whole* path will be enforced: $\sum_i x_i \geq 100$ with i denoting every arc in the path. A value lower than 100 signifies not all arcs are enforced.

[14] Only this once will the full path be given, due to cluttering of the thesis.
[15] These are the 33^{rd} and 34^{th} arcs of both λ_5 and λ_6.
[16] Meaning that there are 4 arcs that are different in λ_5 and λ_6.

λ_5(Value: 0.2381) $= (0 \to 2), (2 \to 4), (4 \to 6), (6 \to 7), (7 \to 10),$
$(10 \to 11), (11 \to 13), (13 \to 16), (16 \to 18), (18 \to 20),$
$(20 \to 22), (22 \to 24), (24 \to 25), (25 \to 27), (27 \to 30),$
$(30 \to 31), (31 \to 34), (34 \to 35), (35 \to 38), (38 \to 39),$
$(39 \to 41), (41 \to 44), (44 \to 46), (46 \to 47), (47 \to 49),$
$(49 \to 52), (52 \to 53), (53 \to 56), (56 \to 57), (57 \to 59),$
$(59 \to 62), (62 \to 63), (63 \to 66), (66 \to 67), (67 \to 69),$
$(69 \to 71), (71 \to 73), (73 \to 75), (75 \to 78), (78 \to 79),$
$(79 \to 82), (82 \to 84), (84 \to 86), (86 \to 87), (87 \to 89),$
$(89 \to 91), (91 \to 93), (93 \to 95), (95 \to 97), (97 \to 100),$
$(100 \to 101), (101 \to 103), (103 \to 105), (105 \to 108),$
$(108 \to 110), (110 \to 111), (111 \to 113), (113 \to 115),$
$(115 \to 117), (117 \to 120), (120 \to 121), (121 \to 124),$
$(124 \to 126), (126 \to 128), (128 \to 130), (130 \to 132),$
$(132 \to 133), (133 \to 136), (136 \to 138), (138 \to 140),$
$(140 \to 141), (141 \to 143), (143 \to 145), (145 \to 147),$
$(147 \to 149), (149 \to 151), (151 \to 154), (154 \to 156),$
$(156 \to 158), (158 \to 160), (160 \to 162), (162 \to 164),$
$(164 \to 165), (165 \to 168), (168 \to 170), (170 \to 171),$
$(171 \to 174), (174 \to 175), (175 \to 177), (177 \to 180),$
$(180 \to 181), (181 \to 184), (184 \to 186), (186 \to 188),$
$(188 \to 189), (189 \to 191), (191 \to 193), (193 \to 196),$
$(196 \to 197), (197 \to 200), (200 \to 201)$

λ_6(Value: 0.7619) $= (0 \to 2), (2 \to 4), (4 \to 6), (6 \to 7), (7 \to 10),$
$(10 \to 11), (11 \to 13), (13 \to 16), (16 \to 18), (18 \to 20),$
$(20 \to 22), (22 \to 24), (24 \to 25), (25 \to 27), (27 \to 30),$
$(30 \to 31), (31 \to 34), (34 \to 35), (35 \to 38), (38 \to 39),$
$(39 \to 41), (41 \to 44), (44 \to 46), (46 \to 47), (47 \to 49),$
$(49 \to 52), (52 \to 53), (53 \to 56), (56 \to 57), (57 \to 59),$
$(59 \to 62), (62 \to 63), (63 \to 65), (65 \to 68), (68 \to 70),$
$(70 \to 71), (71 \to 73), (73 \to 75), (75 \to 78), (78 \to 79),$
$(79 \to 82), (82 \to 84), (84 \to 86), (86 \to 87), (87 \to 89),$
$(89 \to 91), (91 \to 93), (93 \to 95), (95 \to 97), (97 \to 100),$
$(100 \to 101), (101 \to 103), (103 \to 105), (105 \to 108),$
$(108 \to 110), (110 \to 111), (111 \to 113), (113 \to 115),$
$(115 \to 117), (117 \to 120), (120 \to 121), (121 \to 124),$
$(124 \to 126), (126 \to 128), (128 \to 130), (130 \to 132),$
$(132 \to 133), (133 \to 136), (136 \to 138), (138 \to 140),$
$(140 \to 141), (141 \to 143), (143 \to 145), (145 \to 147),$
$(147 \to 149), (149 \to 151), (151 \to 154), (154 \to 156),$
$(156 \to 158), (158 \to 160), (160 \to 162), (162 \to 164),$
$(164 \to 165), (165 \to 168), (168 \to 170), (170 \to 171),$
$(171 \to 174), (174 \to 175), (175 \to 177), (177 \to 180),$
$(180 \to 181), (181 \to 184), (184 \to 186), (186 \to 188),$
$(188 \to 189), (189 \to 191), (191 \to 193), (193 \to 196),$
$(196 \to 197), (197 \to 200), (200 \to 201)$

6.2.1 Measuring the computation time

In order to get reliable results, these solutions will be calculated 50 times, in an attempt to prevent seemingly random variation of the calculation times: a multitasking OS, like Windows (on which the code was run), shares the CPU among different processes, which can affect the calculation time. Although this precaution is meant to avoid fluctuations, the results will still only give a rough estimation of the relative calculation time difference between the different branching tactics.
Table 4 displays the duration per iteration (in seconds) for each strategy, which helps determine the approach requiring the least calculation time. This determination is based on the average calculation time over 50 runs. The table caption further provides the means of the three tactics' measurements and the error on those means. This error was calculated[17] with the standard error formula $SE = \frac{\sigma}{\sqrt{N}}$, with $\sigma = \sqrt{\frac{\sum_i (x_i - \mu)^2}{N}}$ the standard deviation, N the sample size (50), μ the mean and x_i each iteration's time.

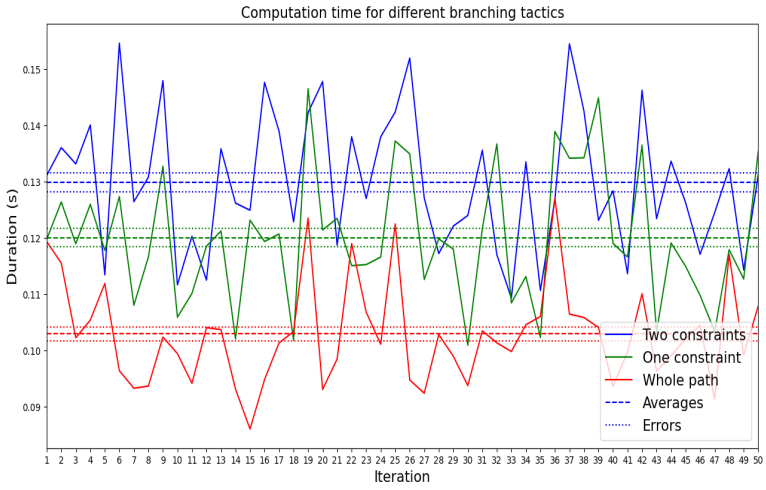

Figure 9: A plot of the three branching decisions and their respective computation time when enforcing the arcs/path with the highest flow.

Figure 9 displays the calculation times noted in Table 4 for each iteration, along with horizontal lines indicating the mean time and error on that mean.
It is apparent that branching with two constraints takes relatively long to calculate, compared to adding multiple arcs into one constraint. Branching on the whole path is relatively faster than branching on two arcs at once.

[17] Using the Python "scipy" module

#	One constr.	Two constr.	Whole path	#	One constr.	Two constr.	Whole path
1	0.109040	0.103459	0.082661	26	0.140678	0.148940	0.105118
2	0.101789	0.109793	0.106308	27	0.126013	0.140562	0.119407
3	0.112985	0.113617	0.093330	28	0.131530	0.125194	0.110436
4	0.121692	0.114001	0.107839	29	0.110230	0.112472	0.093870
5	0.100758	0.117572	0.092544	30	0.127623	0.133796	0.093015
6	0.115355	0.107354	0.100253	31	0.099430	0.116863	0.086294
7	0.135989	0.117676	0.115001	32	0.113696	0.115268	0.093946
8	0.121727	0.127453	0.115332	33	0.093540	0.123803	0.094508
9	0.117693	0.136732	0.100267	34	0.125037	0.122009	0.101531
10	0.122367	0.120851	0.117561	35	0.120037	0.128525	0.114355
11	0.118999	0.122994	0.107697	36	0.111370	0.132872	0.111403
12	0.125546	0.143545	0.109883	37	0.113715	0.144372	0.099323
13	0.125299	0.125547	0.114521	38	0.115058	0.107288	0.101332
14	0.119052	0.123980	0.089508	39	0.104135	0.115021	0.100569
15	0.106079	0.111046	0.102198	40	0.131046	0.125658	0.113385
16	0.124712	0.114506	0.090977	41	0.118274	0.122456	0.099477
17	0.119446	0.116896	0.097580	42	0.103519	0.112463	0.103021
18	0.113258	0.123915	0.099525	43	0.099186	0.114423	0.090082
19	0.104170	0.138934	0.111202	44	0.119209	0.119920	0.116435
20	0.131624	0.120370	0.102590	45	0.115075	0.130814	0.102732
21	0.117150	0.124067	0.101949	46	0.108635	0.108555	0.107070
22	0.127853	0.110692	0.080597	47	0.102453	0.115419	0.087978
23	0.112924	0.127292	0.096714	48	0.110547	0.132815	0.106097
24	0.109763	0.118561	0.100729	49	0.122954	0.110618	0.112161
25	0.098525	0.123035	0.112553	50	0.101562	0.116595	0.100894

Table 4: All 50 calculation times (in seconds) per branching tactic when enforcing (some arcs from) λ_6. The mean and the error on the mean was also calculated. These were, respectively, for the one-constraint, two-constraint and whole path-constraint (0.120010 ± 0.001637)s, (0.129897 ± 0.001709)s and (0.102898 ± 0.001274)s.

In order to obtain more information about the computation times, some other settings can be investigated:

- Table 4 shows the computation times for different branching tactics when enforcing whole, or parts of path λ_6 (the path with the highest flow in the first column generation solution).
  There are three other possible approaches:

  1. Omitting the arcs of λ_6 that were previously enforced (i.e. changing "≥ 1" into "$= 0$").
  2. Enforcing the corresponding arcs from λ_6 in path λ_5. For clarification: in λ_6, the branching arcs are $(63 \to 65)$ and $(65 \to 68)$; in λ_5, the corresponding arcs would be $(63 \to 66)$ and $(66 \to 67)$.
  3. The last possible approach would be to omit the same corresponding arcs from λ_6 in path λ_5.

- It could be worthwhile to examine what happens when the constraint

value enforcing the whole path (currently 100) is lowered. For instance, comparing the times for the constraint values 99, 98, 97 and 96. Each value enforces fewer and fewer arcs, meaning a value of 96 won't strictly enforce any arcs with non-integer flows (as all there are 96 arcs that already have a flow of 1). To compare, the constraint values 50 and 1 will also be used; a hypothesis is that these last two constraint values will have the same computation time as constraint value 96 (as no non-integer arcs are strictly enforced).

Different branching constraints Figure 10 shows the effect of changing the constraint values from "$= 1$", "≥ 2" and "≥ 100" to "$= 0$" for (the branching arcs in) λ_6. This change makes it so the branching arcs and the whole path of λ_6 are omitted.
Omitting the whole path with the highest flow has the highest average calculation time, while the one-constraint and two-constraints times are relatively fast and similar.

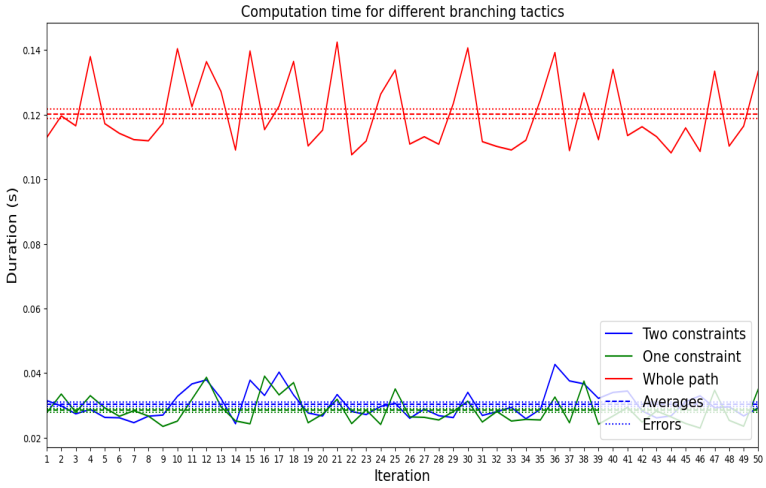

Figure 10: A plot of the three branching decisions and their respective computation time when omitting the arcs/path with the highest flow.

One could also branch on the fractional arcs in λ_5. Figure 11 shows how the computation times associated with the three branching tactics compare for a constraint value of "$= 0$". Omitting the whole path is relatively slower than constraining only 2 arcs. Additionally, the one-constraint time is now nearly the same as the two-constraints time.

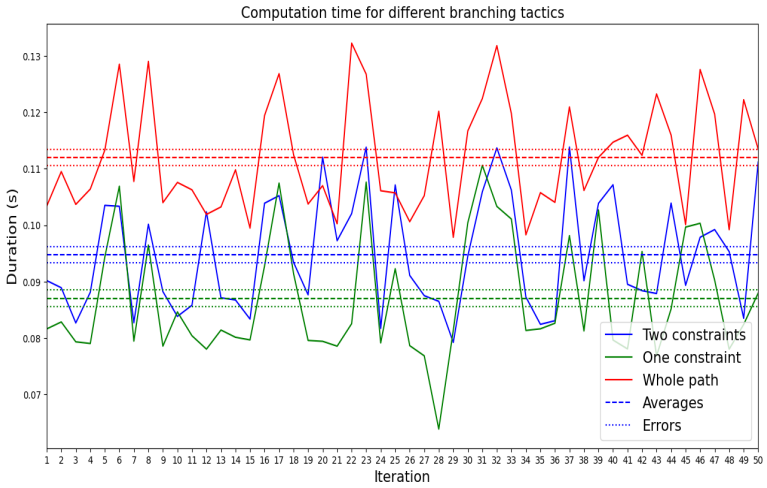

Figure 11: A plot of the three branching decisions and their respective computation time when omitting the arcs/path with the lowest flow.

Lastly, the (fractional) arcs of λ_5 can be enforced. This means the arcs/path with the lowest flow are to be in the solution. Figure 12 displays how, just as when omitting λ_5, branching the whole path takes the longest, while the two-arcs branches are similar in computation time.

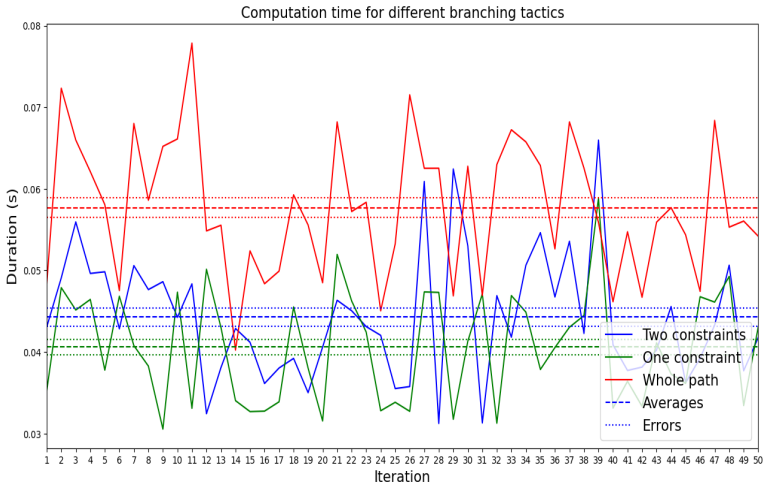

Figure 12: A plot of the three branching decisions and their respective computation time when enforcing the arcs/path with the lowest flow.

To make this information more clear, Table 5 summarizes the relations found in Figures 9-12.

Path	Enforced	Omitted
λ_5	Constraining the whole path takes the longest, while the other two branching tactics are similar in time	Constraining the whole path takes the longest, while the other two branching tactics are similar in time
λ_6	Using multiple constraints is the slowest; branching on the whole path is the fastest	Constraining the whole path takes the longest, while the other two branching tactics are similar in time

Table 5: A comparison between the different branching tactics and constraint values, where the flow of $\lambda_6 > \lambda_5$.

Different path constraint values Another way to vary the branching is to alter the value enforcing a path. To be more precise, the previous constraint enforcing a whole path was "≥ 100", but it will now be lowered. A hypothesis of the effect this has on the computation time is that it lowers as the amount of constrained arcs is lowered. Specifically, there are 4 arcs being enforced if the constraint value is "≥ 100". Because 96 of the 100 arcs have an integer flow, if the value drops to "≤ 96", the same path will likely be found as in the first column generation (while possibly needing the same amount of time to calculate the solution). Figures 13, 14, 15 and 16 respectively show how the computation time of branching the "whole" path changes as the constraint value is lowered from "≥ 99" to "≥ 96" (with the same settings as in Figure 9). Figures 17 and 18 are respectively the computational time plot for values of "≥ 50" and "≥ 1".

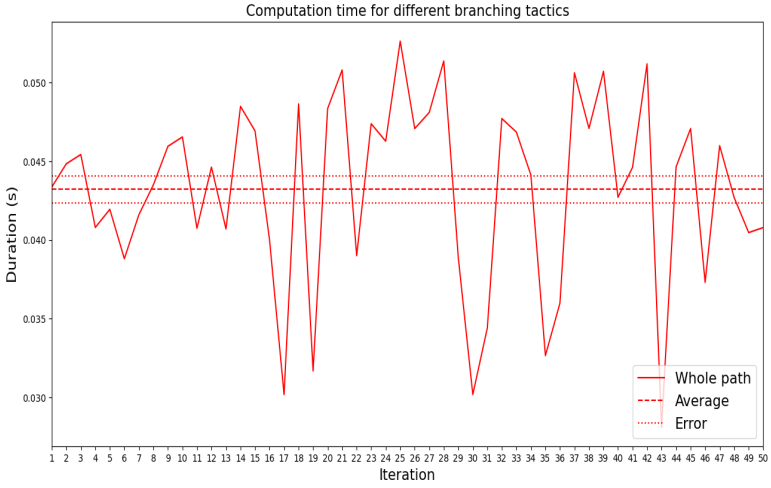

Figure 13: A plot of the computation time when enforcing the whole path with the highest flow. The path-enforcing constraint has a value of "≥ 99".

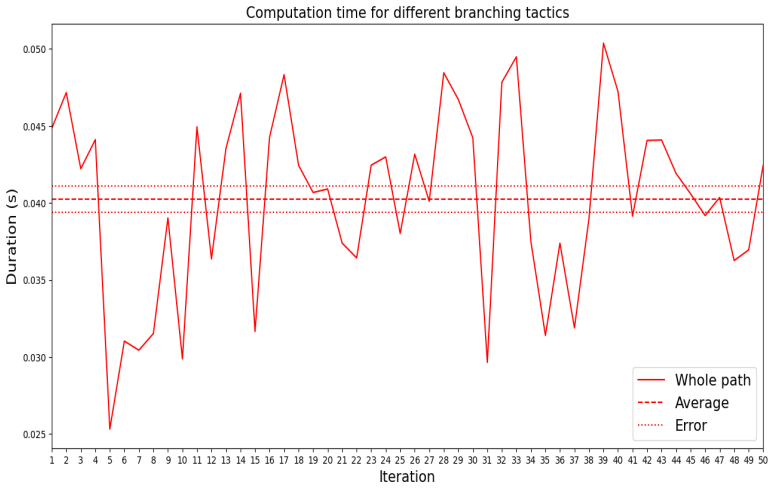

Figure 14: A plot of the three branching decisions and their respective computation time when enforcing the arcs/path with the highest flow. The path-enforcing constraint has a value of "≥ 98".

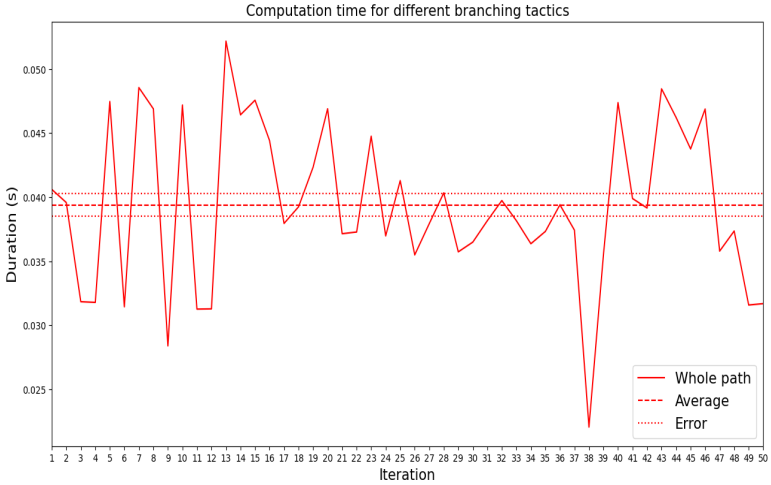

Figure 15: A plot of the three branching decisions and their respective computation time when enforcing the arcs/path with the highest flow. The path-enforcing constraint has a value of "≥ 97".

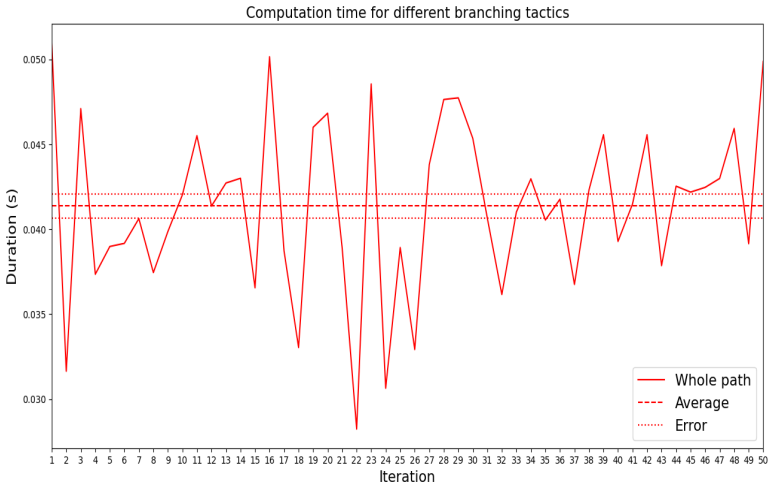

Figure 16: A plot of the three branching decisions and their respective computation time when enforcing the arcs/path with the highest flow. The path-enforcing constraint has a value of "≥ 96".

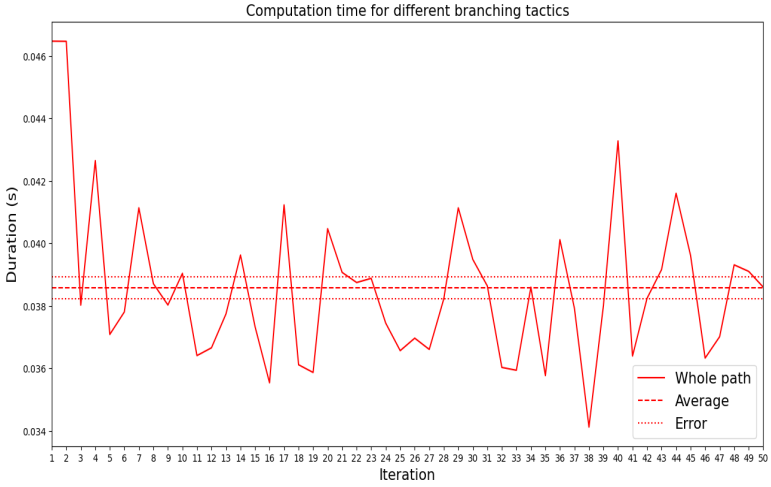

Figure 17: A plot of the three branching decisions and their respective computation time when enforcing the arcs/path with the highest flow. The path-enforcing constraint has a value of "≥ 50".

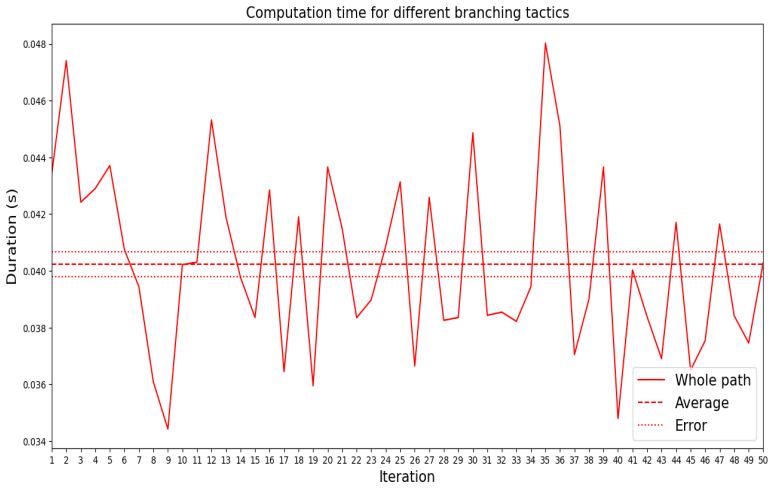

Figure 18: A plot of the three branching decisions and their respective computation time when enforcing the arcs/path with the highest flow. The path-enforcing constraint has a value of "≥ 1".

These measurements all seem to have similar computation times. The means of these computation times are about half of the mean found in Figure 9, when the constraint value was "≥ 100". When strictly enforcing all non-integer arcs in λ_6, the computation time is relatively low compared to the other two tactics. However, when the sum of the flow of these arcs is not forced to be "1", the computation time drops.
All constraint values < 100 produce about the same computation time. An explanation for this could be that the sum of the non-integer flows of these 4 arcs is already > 3 in λ_6: the flow of each of these arcs is equal to 0.7619. Seen as $4 \cdot 0.7619 = 3.0476$, the constraint value of "≥ 99" is already satisfied; there are 96 arcs with flow "1", and 4 arcs with flow "0.7619", which means that the sum of flows of all arcs in λ_6 equals $96 \cdot 1 + 4 \cdot 0.7619 = 99.0476$.

7 Conclusion

In this thesis, a broad theoretical background was given about Linear Programs (LPs), the Simplex method, column generation and Branch-and-Bound. These subjects have many real-world use-cases, from logistics and finance to manufacturing and network design.

An LP can either be minimized or maximized, referring to the objective function, which is subject to the LP's variables and constraints. The Simplex method finds the optimal solution by iteratively moving along the edges of the solution space's feasible region, from vertex to vertex. The column generation algorithm is especially useful when dealing with very large LP problems: explicitly formulating such a large LP would require enumerating all possible paths, which is

impractical. This algorithm is innovative in the sense that its variables are now whole paths; additionally, it only introduces the most relevant variables into the model, as the Restricted Master Problem (RMP) (containing only a subset of variables) iteratively gets solved.
The solutions of column generation can be non-integer, while many real-world applications require integer solutions. Although the optimal path is a combination of multiple fractional paths, a flight scheduler, for example, will not find this solution to be very useful. Branch-and-Bound provides a way to get integer solutions by adding constraints to the LP.

The majority of the time coding for this thesis was spent on implementing the example in [2]. Firstly, by understanding how to program the Simplex method, and adding an artificial variable, so a feasible starting solution can always be found if the LP itself is feasible. Secondly, the column generation algorithm was implemented by creating a subproblem function which can be called in the iteration-loop. The subproblem uses the dual variables obtained from solving the RMP and determines the new best path. Ultimately, the Python code reaches the same solutions as in the example it originally tries to implement.

A first small-scale research was done on three different branching tactics tried to determine the relative ranking/difference between their calculation times. To do this, a large LP was constructed of 202 nodes and 400 arcs. An initial solution was found using column generation: a path (λ_6) with flow 0.7619 and a path λ_5 with a lower flow 0.2381. Only 8 arcs contained a non-integer flow (meaning there are four arcs different in both paths).
The results, although skewed by random fluctuations within computer software, convey how the computation time alters when either enforcing or omitting the (non-integer) arcs of one of the paths found in the initial column generation (being λ_5 and λ_6). By choosing two arcs with non-integer flow from λ_6, two of the three branching tactics can be constructed: one of these puts the two arcs in one constraint, the other separates the arcs into two constraints. The third branching tactic was to put every arc in the path into the same constraint, including the ones with integer arc flows. The same three branching tactics were used on the corresponding arcs of λ_5.
The computation times of enforcing and omitting arcs were examined for all three branching tactics: the solution of each tactic was calculated 50 times. Afterwards, the mean times were determined, which were then used to describe the slowest and fastest branching tactic per constraint type. Enforcing arcs in the path with the highest flow, λ_6, gave a unique result: branching on the whole path is fastest, and using multiple constraints is the slowest. All three other constraining types (omitting arcs in λ_6 and λ_5, and enforcing arcs in λ_5) resulted in the same conclusion: branching on the whole path is the slowest, while the two other branching tactics are similar in computation time.
The second research examined the computation times when altering the constraint value enforcing the path. Previously, when a path was enforced, *all* its arcs were enforced, meaning the sum of the flow through every arc had to be "≥ 100" (because there are 100 arcs in the path). In this case, that value is lowered: the values "≥ 99", "≥ 98", "≥ 97", "≥ 96", "≥ 50" and "≥ 1" were examined. Each time a value is lowered, fewer arcs are strictly enforced; i.e. when the constraint value is "≥ 99", there could be one arc with a flow of 0.8 (result-

ing in a sum of 99.8), but it could also be the case that two arcs have a flow of 0.8, meaning the sum of flows is 99.6. This is exactly what happened when the constraint value was lowered from "≥ 100" to "≥ 99": the four non-integer arcs in λ_6 each have a flow of 0.7619. Their combined flow is $4 \cdot 0.7619 = 3.0476$, meaning the constraint "≥ 99" is already satisfied. This is why the computation times for every constraint value lower than 100 are similar; they are also nearly half the time that was measured when the constraint value was "≥ 100".

Additional works could build on this thesis' code and primitive findings to fully the Branch-and-Bound algorithm and investigate the branching trees: it could be interesting to examine the amount of branches necessary to converge to an integer solution.

Appendix

The code used to solve the Primer's model ([2]) in Section 6.1 can be found using the following link:
`https://github.com/DipiDopo/Bachelor-s-thesis-Alexander-Colman`.
This is a GitHub repo containing a README, giving an overview of the repo, alongside two folders. The folder "READMEs" contains three .md files, which describe the contents of the code files, found in the folder "CODE". These .md files (and the comments in the .py files themselves) try to give a clear understanding of how and why the code was written.

The main file, implementation.py, holds a class ColumnGeneration. This class has multiple methods, one of which is called "column_generation", which will determine the optimal solution of a graph using the column generation algorithm. Besides the class, the file also contains a large number of functions, which can be called to run specific scenarios; five of these functions directly run the five Branch-and-Bound iterations from the Primer.
runner.py can be used to easily run the functions created in implementation.py; all variables are already assigned and given a value.
research.py also uses implementation.py, but creates its own graph, as was explained in Section 6.2. After generating the graph, it runs the column generation process (with branching) 50 times for every branching tactic.

References

[1] Van Den Steen, Y (2023) *Combining the interior point method with column generation for linear programming.* MA thesis. University of Antwerp.

[2] Desrosiers, J., Lübbecke, M.E. (2005). *A Primer in Column Generation.* In: Desaulniers, G., Desrosiers, J., Solomon, M.M. (eds) Column Generation. Springer, Boston, MA.

[3] Bazaraa, M. S., Jarvis, J. J., & Sherali, H. D. (2010). *Linear Programming and Network Flows.* (4th ed.). John Wiley & Sons.

[4] Silva, R. (2018). *WRENCH-Pegasus Implementation: Montage-1000-001.* GitHub. `https://github.com/wrench-project/pegasus/blob/master/examples/evaluation/scalability/montage-1000-001.xml`. Visited on 16/04/2025.

[5] Bertsimas, D. and Tsitsiklis, J.N. (1997). *Introduction to Linear Optimization.* MIT.

[6] *Geogebra calculator* (z.d.). GeoGebra. `https://www.geogebra.org/calculator`. Visited on 07/05/2025.

[7] Nering, E. D., & Tucker, A. W. (1993). *Linear Programs and Related Problems.* ACADEMIC PRESS, INC.

[8] Purdue University Northwest (z.d.). *Linear Programming: the Simplex Method.* In LECTURE NOTES (pp. 70–75).

[9] Dantzig, G. B. (1963). *Linear Programming and Extensions.* Princeton University Press.

[10] Rousseau, L.-M. (2016). *Introduction to Column Generation.* `https://symposia.cirrelt.ca/system/documents/000/000/254/Rousseau_original.pdf?1464701234`. Visited on 29/03/2025.

[11] Pal, S. (2021). *Column Generation.* [Thesis]. Department of Computer Science and Engineering, Indian Institute of Technology, Bombay.

[12] Winston, W. L. (z.d.). *Integer Programming: The Branch and Bound Method.* Instituto Superior Técnico. `https://web.tecnico.ulisboa.pt/mcasquilho/compute/_linpro/TaylorB_module_c.pdf`. Visited on 01/05/2025.