CHALMERS
UNIVERSITY OF TECHNOLOGY

An Improved Shortest Path Algorithm for Aircraft Routing

A Comparison of Dynamic Programming Algorithms

Master's thesis in Engineering Mathematics and Computational Science

SAMUEL THORÉN

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden 2022
www.chalmers.se

Master's thesis 2022

An Improved Shortest Path Algorithm for Aircraft Routing

A Comparison of Dynamic Programming Algorithms

Samuel Thorén

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology
Gothenburg, Sweden 2022

An Improved Shortest Path Algorithm for Aircraft Routing
A Comparison of Dynamic Programming Algorithms
Samuel Thorén

Supervisor: Mattias Grönkvist, Jeppesen
Supervisor: Ann-Brith Strömberg, Department of Mathematical Sciences, Chalmers University of Technology
Examiner: Ann-Brith Strömberg, Department of Mathematical Sciences, Chalmers University of Technology

Master's Thesis 2022
Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2022

An Improved Shortest Path Algorithm for Aircraft Routing
A Comparison of Dynamic Programming Algorithms
Samuel Thorén
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

Thousands of commercial flights are scheduled each day, and the need for flight routes to be as efficient as possible is very important. The cost for the flights has to be as low as possible while still fulfilling and complying with all regulations in safety and other requirements.

Aircraft routing is the problem of creating routes for aircraft to operate, subject to constraints. The shortest path problem with resource constraints is commonly used as a subproblem in solution methodologies in order to solve the aircraft routing problem. The aim of this thesis is to implement and evaluate an algorithm called a multidirectional dynamic programming algorithm (MDDPA) in order to solve the shortest path problem with resource constraints. The results from the implementation of MDDPA is compared to the results of another dynamic programming algorithm.

The results show that MDDPA in most test cases finds a solution with a cost that is at least as good as the solution found by the other dynamic programming algorithm. However, this usually comes at the expense of longer execution times, where our implementation of MDDPA in most cases takes longer time at finding a solution. In order to gain better results in terms of computing times, we suggest implementing multithreading in MDDPA and analyse how the choice of parameters used in MDDPA could affect the results.

Keywords: Multidirectional dynamic programming algorithm (MDDPA), Dynamic programming, Shortest path problem with resource constraints, Aircraft routing, Column generation

Acknowledgements

First and foremost I want to thank Jeppesen for giving me this opportunity for writing my master's thesis with them. I want to thank my supervisor Mattias Grönkvist at Jeppesen for all the discussions and help with the thesis. And also a big thank you to everyone I met during my time at Jeppesen for making this a wonderful experience.

Thanks too Ann-Brith Strömberg at Chalmers for examining this project. Also thanks to Lukas Enarsson and Karthikeyan Jeganathan for giving me feedback on my thesis. I also want to thank Anna Andersson and Axel Thorén for help with proofreading of this thesis.

Samuel Thorén, Gothenburg, June 2022

List of Acronyms

Listed below are the acronyms that have been used, listed in alphabetical order:

FDD	Feasible descent direction
IDD	Improving descent direction
LLEL	Learning from locally efficient labels
LLS	Label loading strategy
LSP	Label storing procedure
MDDPA	Multidirectional dynamic programming algorithm
NF	Nearest first
SPPRC	Shortest path problem with resource constraints

Table of Contents

1 Introduction

In 2010 there were a total of 27.8 million commercial flights performed by the global airline industry. This number has increased by about 1 million flights per year up until the COVID-19 pandemic started. In 2021 there were only 19.3 million commercial flights performed by the global airline industry, but it is rapidly increasing again [1]. Due to the large amount of flights, there is a lot of planning involved to make the aircraft planning process as efficient as possible. In order to minimise costs and improve profitability for the airlines, there is a lot of interest in making the plans for the aircraft routes more efficient. This can be done in a lot of ways, for instance by making the crew planning more efficient or by making more efficient routes for the flights. The airlines must still ensure safety for the travellers and increase customer satisfaction, and have to take for instance aircraft maintenance and the time for layovers into consideration.

Jeppesen in Gothenburg work with optimising aircraft routing for their customers using optimising algorithms. Aircraft routing is the problem of creating routes for aircraft to operate, subject to constraints. These constraints could for instance concern airport curfews, aircraft restrictions, and requirements and planning for aircraft maintenance. The shortest path problem with resource constraints (SPPRC) is commonly used as subproblem in many solution methodologies for aircraft routing, for instance inside column generation. SPPRC is an optimisation problem that consists of finding the shortest path between two nodes in a network in such a way that some cost in minimised, with respect to resource constraints [2].

Right now, Jeppesen uses column generation with an integer fixing heuristic in order to solve the aircraft routing problem. Even though this way of solving the problem works well, Jeppesen is interested in investigating if the solution times and the best feasible solutions found could be improved. To do this, an algorithm called a multidirectional dynamic programming algorithm (MDDPA) for the SPPRC is going to be implemented. This algorithm

is described by Ilyas Himmich, Issmail El Hallaoui & François Soumis [3]. MDDPA is going to be implemented in `C++` and tested on real-world data from different airlines that are customers to Jeppesen in order to evaluate and compare MDDPA to the currently used algorithm that Jeppesen use. Comparisons between the total execution times and the costs of the best feasible paths found are going to be carried out.

1.1 Aim

The aim is to implement MDDPA in `C++` in order to evaluate and compare the results for the SPPRC between MDDPA and the currently used algorithm at Jeppesen. Comparisons between the total execution times for running the algorithm and the costs of the best feasible paths found are going to be carried out between the two algorithms. The aim is also to compare the results for different types of problems, does MDDPA give better results for some types of problems?

1.2 Demarcations

There might be other algorithms that could be implemented and tested in order to solve the problem for the SPPRC, but for this project the only algorithm that is going to be tested is MDDPA. In the article describing MDDPA [3], the authors describe two different strategies that could be implemented for the *label loading strategy*. Only one of these strategies is going to be implemented, namely the one called *nearest first*. This is due to time constraints, and also *nearest first* is better suited for implementation with the current code at Jeppesen.

1.3 Ethical aspects

The ethical aspects that are taken into consideration are mostly the environmental benefits. If the results are promising and the computational times are lower, then most likely the required energy for the computations will decrease, which is beneficial for the environment. Promising results could also lead to the airlines using their aircraft more efficiently which could reduce the amount of fuel consumption, which is also an environmental benefit.

2 | Theory

This chapter presents all relevant theory. The first sections describe the tail assignment problem and methods used in order to obtain the pricing problem, and how Jeppesen solves the SPPRC with the algorithm they currently use. The later sections describe MDDPA in detail and the data used to test and evaluate the implementation of MDDPA.

2.1 Tail assignment problem

Aircraft routing is the most common variant of aircraft assignment and the tail assignment problem is a type of aircraft assignment problem, which covers the entire aircraft planning process and captures all operational constraints. These constraints could be a number of things, such as maintenance requirements and planning, airport curfews, aircraft restrictions, and connection times. The aim of the tail assignment problem is to find a solution where all flights are assigned to an aircraft in such a way that all constraints are satisfied, in addition to minimising some cost function [2]. The tail assignment problem is defined in Model 1, where (2.1a) is the function to minimise and (2.1b)-(2.1g) are the constraints.

For the tail assignment problem described in Model 1, there are five different sets. In this model, T is the set of aircraft that are going to be planned. F is the set of activities, which can for instance be maintenance or ground activities, individual flight legs, or a sequence of flights. The set P_t is the set of activities that are preassigned to an aircraft t, and R_t is the set of activities that an aircraft t is not allowed to operate. Lastly the set M denotes the maintenance types.

Model 1. *The tail assignment problem.*

$$\text{minimise} \quad \sum_{i\in F}\sum_{j\in F}\sum_{t\in T} c_{ijt}y_{ijt} \tag{2.1a}$$

$$\text{subject to} \quad \sum_{j\in F} y_{jit} - \sum_{j\in F} y_{ijt} = 0, \quad \forall t \in T,\ \forall i \in F, \tag{2.1b}$$

$$\sum_{t\in T}\sum_{j\in F} y_{ijt} = 1, \quad \forall i \in F, \tag{2.1c}$$

$$\sum_{j\in F} y_{ijt} = 1, \quad \forall i \in P_t,\ \forall t \in T, \tag{2.1d}$$

$$\sum_{j\in F} y_{ijt} = 0, \quad \forall i \in R_t,\ \forall t \in T, \tag{2.1e}$$

$$r_{im} \leq l_m, \quad \forall i \in F,\ \forall m \in M, \tag{2.1f}$$

$$y_{ijt} \in \{0,1\}, \quad i,j \in F,\ \forall t \in T \tag{2.1g}$$

The objective function (2.1a) describes the function that is going to be minimised. The variable y_{ijt} is the binary decision variable and is equal to 1 if there is a connection between activity i and j for aircraft t. Similarly, the cost c_{ijt} is the cost for the connection between activity i and j for aircraft t. The objective is to minimise the sum of all the connections between the activities for all of the aircraft.

The first constraints, (2.1b), are ensuring that if there is a connection to an activity, there must also be a connection from the activity. Constraints (2.1c) ensure that all of the activities are covered exactly once each. Constraints (2.1d) ensure that all preassigned activities are carried out, and constraints (2.1e) ensure that the activities that aircraft t is not allowed to operate are not carried out by aircraft t. Constraints (2.1f) are resource constraints and can for instance be that an aircraft t gets enough maintenance, or that it is not exceeding a number of flying hours. Note that in Model 1 the variables r_{im} and y_{ijt} are not connected. In the implementation they are, however, interacting via recursive constraints that are complicated to describe in a mixed-integer linear model. The last constraints (2.1g) ensure that the decision variables are binary.

Model 1 can be rewritten as Model 2, which is a set partitioning model. This is done using path flows, with the use of the flow decomposition theorem (see [4]). Path flows are defined in Definition 1 (see [5]). The reason for rewriting the model is that using column generation on Model 2 works very well, and also that constraints (2.1f) are recursive constraints, which are hard to handle in a linear model (see [2], Ch. 4).

Model 2. *The tail assignment problem as a set partitioning problem.*

$$\text{minimise} \quad \sum_{r \in R} c_r x_r \tag{2.2a}$$

$$\text{subject to} \quad \sum_{r \in R} a_{fr} x_r = 1, \quad \forall f \in F, \tag{2.2b}$$

$$\sum_{r \in R} b_{qr} x_r \leq d_q, \quad \forall q \in Q, \tag{2.2c}$$

$$x_r \in \{0, 1\} \tag{2.2d}$$

In Model 2, the set R is the set of all possible paths which corresponds to the flows that satisfies all of the individual route constraints. The set F is still the set of all activities. The decision variable x_r is a binary variable and is equal to 1 if route r is used, otherwise equal to 0. The objective function is still to minimise the sum of the total cost, as seen in (2.2a). In the first constraint, (2.2b), a_{fr} is equal to 1 if activity f is covered by route r, otherwise a_{fr} is equal to 0. This constraint ensures that all activities are covered once. The constraints (2.2c) shows the global constraints, where b_{qr} is the contribution to the global constraint q from route r. A global constraint is a type of resource constraint that holds for all aircraft in a problem. Here, d_q is a value for global constraint q, and Q is the set of all global constraints. For most problems, the set Q is going to be empty. The last constraint ensures that the decision variable is binary.

Definition 1. *For a path P in a network, a path flow $f(P)$ is a flow with the property such that there is a positive number k, such that $f(P)_{ij} = k$ if the arc from node i to j is an arc of P, otherwise $f(P)_{ij} = 0$.*

2.2 Dantzig-Wolfe decomposition and column generation

Since the problem in Model 2 has exponentially many variables and thus is very large, the method used to solve the model is Dantzig-Wolfe decomposition and column generation. Dantzig-Wolfe decomposition was first described by George B. Dantzig and Philip Wolfe in [6], and can be used to solve a problem in the form of Model 3.

Model 3. *Linear programming problem.*

$$\text{minimise} \quad \sum_{j=1}^{n} c_j x_j \tag{2.3a}$$

$$\text{subject to} \quad \sum_{j=1}^{n} A_j x_j \leq b_0, \tag{2.3b}$$

$$B_j x_j \leq b_j, \forall j, \tag{2.3c}$$

$$x_j \geq 0 \tag{2.3d}$$

The objective function in Model 3 is to minimise the linear function in (2.3a), where c_j are the costs, and $x_j \geq 0$ is a decision variable where $j = (1, \ldots, n)$. The constraints in (2.3c) can be seen as defining n different subproblems, and for the sake of simplicity the convex polyhedron S_j is assumed to be non-empty and bounded for $j = (1, \ldots, n)$ and is defined as

$$S_j = \{x_j | x_j \geq 0, B_j x_j \leq b_j\}.$$

The variable x_j can be written as a convex combination of the j^{th} component of the extreme points in S_j (see [7], p. 281) such that

$$\bar{x}_j = \sum_{i=1}^{L_j} \lambda_j^i x_j^i,$$

where L_j is the number of extreme points in the set S_j, x_j^i is the extreme points in S_j, and λ_j^i are parameters such that

$$\sum_{i=1}^{L_j} \lambda_j^i = 1, \quad \forall j = (1, \ldots, n),$$

$$\lambda_j^i \geq 0, \quad \forall i, j.$$

Model 3 can now be rewritten as Model 4 by changing x_j to $\bar{x}_j$.

Model 4. *Linear programming problem, when changing x_j to $\bar{x}_j$ where λ_j^i is the decision variable.*

$$\text{minimise} \quad \sum_{j=1}^{n} \sum_{i=1}^{L_j} c_j x_j^i \lambda_j^i \tag{2.4a}$$

$$\text{subject to} \quad \sum_{j=1}^{n} \sum_{i=1}^{L_j} A_j x_j^i \lambda_j^i \leq b, \tag{2.4b}$$

$$\sum_{i=1}^{L_j} \lambda_j^i = 1, \quad \forall j = (1, \ldots, n), \tag{2.4c}$$

$$\lambda_j^i \geq 0, \quad \forall i = (1, \ldots, L_j), j = (1, \ldots, n) \tag{2.4d}$$

It is clear that Model 4 has fewer constraints than Model 3, but for large problem instances there could still be a lot of constraints. To solve the problem in Model 4, column generation could be applied. Column generation works by having a master problem, in this case Model 4 is set as the master problem. The master problem provides two dual sets of variables: for (2.4b) the dual variable is denoted as u and for the constraints in (2.4c) the dual variables are denoted as $v_j, \ j \in \{1, \ldots, n\}$ [8]. The reduced cost $\bar{c}_j^i$ for variable λ_j^i is given by

$$\bar{c}_j^i = c_j x_j^i - u A_j x_j^i - v_j,$$

where u and v_j are optimal values of the dual variables in a solution to a restricted version of Model 4, where not all variables λ_j^i are generated and $x_j^1, \ldots, x_j^L$ is an extreme point to the set $S_j, \ j = 1, \ldots, n$.

When using column generation, a *pricing problem* is also generated for each j which is shown in Model 5. The objective is to find the minimum possible reduced cost by minimising the function $c_j x_j^i - u A_j x_j^i - v_j$, and if the minimum value is negative it is a reduced cost, where the column where the reduced cost is obtained gets added to the master problem in order to find new dual variables and repeat the process. If no negative reduced cost is found, then the current solution to the master problem is optimal [9].

Model 5. *Pricing problem for each* $j = 1, \ldots, n$.

$$\begin{aligned} \text{minimise} \quad & c_j x_j - u A_j x_j - v_j && \text{(2.5a)} \\ \text{subject to} \quad & B_j x_j \leq b_j, && \text{(2.5b)} \\ & x_j \geq 0 && \text{(2.5c)} \end{aligned}$$

2.3 Pricing problem of the tail assignment problem

Going back to Model 2 and applying Dantzig-Wolfe decomposition [6] and column generation, a reduced cost can be obtained as the following

$$\bar{c}_r = c_r - \sum_{f \in F} a_{fr} u_f - v_r. \tag{2.6}$$

This is the reduced cost for one individual aircraft. In (2.6), u_f is an optimal value of the dual variable for constraint f in (2.2b), and all other variables and parameters are the same as in Model 2. Further, if F_r denotes the set of activities covered by route r — note that a route with minimal reduced cost in this context will be equivalent to a shortest path — (2.6) can be rewritten as

$$\bar{c}_r = c_r - \sum_{f \in F_r} u_f - v_r. \tag{2.7}$$

This can be even further rewritten by assuming $c_r = \sum_{f \in F_r} c_f$, to obtain (2.8) (see [2]).

$$\bar{c}_r = \sum_{f \in F_r} c_f - \sum_{f \in F_r} u_f - v_r = \sum_{f \in F_r} (c_f - u_f) - v_r. \tag{2.8}$$

If a negative reduced cost is found from the pricing problem, the corresponding column gets added to the restricted master problem, which for Model 2 is defined as Model 6.

Model 6. *Restricted master problem for the tail assignment problem using path flows.*

$$\text{minimise} \quad \sum_{r \in R'} c_r x_r \tag{2.9a}$$

$$\text{subject to} \quad \sum_{r \in R'} a_{fr} x_r = 1, \quad \forall f \in F, \tag{2.9b}$$

$$x_r \geq 0, \quad \forall r \in R' \tag{2.9c}$$

In this model, the set $R' \subset R$ denotes the set of routes that are currently available. If no negative reduced cost column is found, then the column generation stops. Each arc in the network has a cost, which means that the pricing problem can be formulated as a shortest path problem [2]. A shortest path problem is the problem of finding the shortest path from a start node to an end node in a graph. For this problem, there are constraints that needs to be taken into consideration, such as maintenance planning and resource consumption. In this case it makes the pricing problem an SPPRC.

2.3.1 Resource constraints for the SPPRC

There are a few different types of constraints for the shortest path problem. These are activity restrictions and preassigned activities, and also cumulative constraints. The first ones are to make sure that each aircraft is not assigned to any activities they are not allowed to do and also that each aircraft gets its preassigned activities. Maintenance for aircraft is very important: every aircraft has to be maintained in regular intervals. This type of maintenance is modelled as preassigned activities, which sometimes can be ongoing for several weeks (see [2]).

There is also another type of maintenance, called *line maintenance*, which is regular maintenance, but the maintenance is carried out with shorter intervals. For instance, this type of maintenance can be carried out after hundred flight hours or after six calendar days. This type of maintenance is modelled with cumulative rules, and ensures that the aircraft has the possibility of being maintained in regular intervals. These constraints turn this problem into an SPPRC. The resources, r_{im}, are accumulated and are not allowed to surpass a certain limit, l_m as expressed in (2.1f) in Model 1. The accumulated resources could however be reset. If there is a chance to make a maintenance check, the resource variables, r_{im}, are reset to zero [2].

There is also a third type of maintenance, but it is not modelled within

the tail assignment problem. This would be corrective maintenance, due to something suddenly breaking, for instance a GPS in an aircraft.

2.4 Currently used algorithm

2.4.1 Label-setting algorithms

The algorithm used for solving the SPPRC is a label-setting algorithm. This means that for each node in the network there is a set of labels. Each label in the set represents a path from the source node to the node the label is at (see [10]). The labels keep track of the cost and the resource constraints. The cost is the accumulated cost of going to the node, in addition to any penalties for using that path. The labels contain a penalty and a running penalty, which are penalties added if a resource constraint were to be exceeded.

Each label also contains a consumption vector, which is a vector that keeps track of the resource consumption of a vehicle. These resource consumptions could be a number of different things, for instance how many hours the aircraft has been flying or how many times the aircraft has landed at a specific airport. The resource consumptions are not allowed to surpass a certain limit.

2.4.2 The labelling algorithm

The currently used algorithm for solving the SPPRC, shown in Algorithm 1, is inspired by the labelling algorithm described by Martin Desrochers and François Soumis [11]. The algorithm starts by initialising a label at the source node with cost and resource constraints set to zero, and an empty set of labels at all other nodes. New labels are created by pulling a label from a node to a new node and thus updating cost and resource consumptions. When pulling a label, if there is a possibility of performing a maintenance check of the aircraft, the resource constraint corresponding to this check will be updated to zero [2].

When pulling labels to a node, all possible labels from the predecessor nodes are taken into account. The labels from the predecessor nodes are temporarily saved and the dominance function, defined in Definition 2, is applied. The labels that are not dominated by any other label are saved: these labels are called efficient labels [2].

Definition 2. *Let l_1 and l_2 be two labels with different feasible partial paths from the source node to node i. Label l_1 is said to dominate l_2 if and only if the cost $c_1 \leq c_2$ and the resource consumptions $r_1^t \leq r_2^t \; \forall t \in T$, and at least one inequality is strict.*

For each node there are two limits, $\underline{\chi}$ and $\bar{\chi}$, such that $\underline{\chi} \leq \bar{\chi}$. Efficient labels are inserted to the node as long as the number of labels is less than $\underline{\chi}$. The labels in the set are sorted lexicographically, meaning that they are first sorted by the reduced cost in increasing order, and then by the resource constraints in increasing order if the reduced cost for two labels are the same. If the added number of labels surpass the lower limit $\underline{\chi}$, only the labels that are lexicographically less than the ones in the set, defined in Definition 3, are added as long as the number of labels at the node do not surpass the upper limit $\bar{\chi}$ [2].

Definition 3. *A label i is lexicographically less than a label j if*

$$\bar{c}_i \leq \bar{c}_j,$$

or if

$$\bar{c}_i = \bar{c}_j \text{ and } \exists l : r_i^l < r_j^l \text{ and } r_i^k = r_j^k \; \forall k \in [0, l),$$

where $\bar{c}_n$ is the reduced cost and r_n^l is the resource consumption at a node n.

The value of the limit $\underline{\chi}$ is chosen in such a way that it is able to balance quality solutions with performance, meaning that the solutions are of good enough quality while still ensuring good enough performance from the algorithm. This will guarantee optimality of the pricing algorithm, however in the context of column generation it is only important to find any label with a negative reduced cost. But in order to find a label with negative reduced cost, if such a label exists, the value for $\underline{\chi}$ must be large enough for the set to contain all of the efficient labels [2]. The value for $\bar{\chi}$ is set in such a way that $\underline{\chi} \leq \bar{\chi}$. The value set for $\bar{\chi}$ is to keep a reasonable amount of labels saved for each node. The use of checking if a label is lexicographically less than another label is a heuristic function used to keep the amount of saved labels at a node at a minimum. If a label is lexicographically larger than another label, it can not dominate the other label and can therefore not be efficient.

When labels have been pulled to the sink node, one or several paths with a negative reduced cost can be obtained if there are any such paths.

Algorithm 1 Currently used algorithm at Jeppesen

```
1: L_{i_source} ← [0, ..., 0]
2: for all i ∈ V \ {i_source} do
3:     L_i ← ∅
4: end for
5: for all i ∈ V \ {i_sink} do
6:     T_i ← L_i
7:     Dominance(T_i)
8:     Sort(L_i)
9:     for each t ∈ T_i do
10:        if size(L_{i+1}) ≤ χ_ then
11:            L_{i+1} ← PullLabel(t)
12:        end if
13:        if size(L_{i+1}) > χ_ and size(L_{i+1}) ≤ χ̄ then
14:            l ← LexicographicallyMaximum(L_{i+1})
15:            if isLexicographicallyLess(t, l) then
16:                L_{i+1} ← Remove(l)
17:                L_{i+1} ← PullLabel(t)
18:            end if
19:        end if
20:    end for
21: end for
```

Algorithm 1 shows the idea behind Jeppesen's algorithm. The functions that are used in this algorithm are `Dominance`, `Sort`, `PullLabel`, `LexicographicallyMaximum`, `isLexicographicallyLess`, `Remove`, and `size`. `Dominance` is the dominance function which is described in Definition 2. `Sort` sorts the labels in lexicographical order, as described in Definition 3. The function `LexicographicallyMaximum` returns the last label in the lexicographically sorted list of labels and `isLexicographicallyLess` is true if the label in the first argument is lexicographically less than the label in the second argument. `PullLabel` refers to the function used to pull the labels to the new nodes. This function also checks the resource constraints and resets them if necessary. The function `Remove` removes a label from a set, and `size` returns the number of labels in a set.

2.5 Multidirectional dynamic programming algorithm

The multidirectional dynamic programming algorithm (MDDPA) is a dynamic programming algorithm that uses labelling [3].

2.5.1 Dynamic programming

Dynamic programming is a method used for solving optimisation problems. It works by breaking down a complicated problem into smaller subproblems and the original problem depends on the solution of these subproblems. By using dynamic programming the subproblems can be split into even smaller problems. These problems can be solved and saved, meaning that they do not have to be solved several times and thus saves computing time [12].

Dynamic programming is used for solving many optimisation problems, and vehicle routing is one of them. There are many different variants of dynamic programming in order to make it more efficient, one of these variants is bi-directional dynamic programming [13]. Bi-directional search is used in bi-directional dynamic programming, meaning that the search states are extended both from the source node and onward, and also from the sink node and backward. The idea behind this is that the paths created from the source node and the sink node can be joined together and create a full path, and thus speeding up the computational time.

Another type of dynamic programming is the MDDPA. The idea behind MDDPA is divided into three steps. The first step is the label storing procedure, which creates small disjoint subspaces by partitioning the state space. These subspaces are then iteratively explored using a label loading strategy. The final step is to more efficiently use the results of previous iterations in order to construct new paths that are promising. The main difference between MDDPA and standard dynamic programming is that MDDPA is able to prove optimality earlier and it performs searches in several directions. [3].

2.5.2 MDDPA

MDDPA is used on an acyclic network $G(V, A)$, with nodes V and directed arcs A. In our case, the networks usually contain multiple vehicles, and the networks can therefore contain multiple source and sink nodes. In MDDPA, only one vehicle with one source node and one sink node will be taken into account at once. MDDPA can therefore be run in parallel for the multiple

vehicles in the network. The algorithm for MDDPA is described in Algorithm 2.

Algorithm 2 Multidirectional dynamic programming algorithm (MDDPA)

1: Define a subnetwork $\bar{G}(\bar{V}, \bar{A})$ of $G(V, A)$, such that $\bar{V} \subset V$ and $\bar{A} \subset A$
2: Compute the reverse shortest path from sink node to source node
3: $c^{best} \leftarrow \infty$
4: **for all** $i \in V \setminus \{i_{source}\}$ **do**
5: $\quad \mathcal{L}_i \leftarrow \emptyset$
6: $\quad \mathcal{S}_i \leftarrow \emptyset$
7: $\quad \mathcal{P}_i \leftarrow \emptyset$
8: **end for**
9: $S \leftarrow \texttt{LSP}(\bar{G}, S)$
10: $k \leftarrow 1$
11: $i^* \leftarrow |V| - 1$
12: **while** $S \neq \emptyset$ **do**
13: $\quad \texttt{LLS}(\mathcal{L}, \mathcal{S}, i^*, k)$
14: $\quad$ **for** $i = i^*$ to i_{sink} **do**
15: $\quad\quad \texttt{Dominance}(\mathcal{L}_i)$
16: $\quad\quad \texttt{LLEL}(\mathcal{L}_i, \mathcal{P}_i, c^{best})$
17: $\quad\quad$ **for all** $(i, j) \in \delta_i^+$ **do**
18: $\quad\quad\quad \mathcal{L}_j \leftarrow \mathcal{L}_j \cup \texttt{Extension}(\mathcal{L}_i, j)$
19: $\quad\quad$ **end for**
20: $\quad\quad \mathcal{P}_i \leftarrow \mathcal{P}_i \cup \mathcal{L}_i$
21: $\quad$ **end for**
22: $\quad \texttt{CostBounding}(c^{best})$
23: $\quad k \leftarrow k + 1$
24: **end while**

The steps 1–11 in MDDPA are the initialisation steps, where the first step is to define a subnetwork $\bar{G}(\bar{V}, \bar{A})$ of $G(V, A)$, where $\bar{V} \subset V$ and $\bar{A} \subset A$. This subnetwork is defined in Section 2.5.4, which describes the function `LSP` where the subnetwork $\bar{G}(\bar{V}, \bar{A})$ is used. The next step is to calculate and save the shortest path from the sink node to each of the nodes in the network. This is calculated using the Bellman-Ford algorithm described in Section 2.5.3, and is used in the function `CostBounding`. Also used in `CostBounding` is the cost of the best path found so far, c^{best}, which is initially set to ∞.

Three sets of labels are then created for each node $i \in V$, the set of active labels $\mathcal{L}_i$, the set of stored labels $\mathcal{S}_i$, and the set of locally efficient labels $\mathcal{P}_i$. The sets for the labels over all nodes can be written as $\mathcal{L} = \cup_{i \in V} \mathcal{L}_i$,

$\mathcal{S} = \cup_{i \in V} \mathcal{S}_i$, and $\mathcal{P} = \cup_{i \in V} \mathcal{P}_i$ respectively. The sets of labels $\mathcal{L}_i$, $\mathcal{S}_i$, and $\mathcal{P}_i$ are all set to the empty set for all nodes except for the source node, where they are set to a label with a cost of zero and all resource consumptions and penalties set to zero. At the end of the initialisation steps the iteration number k is set to one, and the index i^* to the second to last node in the network.

The steps from 12 and onward describe the search procedure, which is in progress as long as there are still stored labels. The search procedure starts with using the function `LLS`, described in Section 2.5.5. The search procedure iterates through i^* to the sink node and applies the dominance function, defined in Definition 2, for the set of active labels $\mathcal{L}_i$. It also applies `LLEL`, defined in Section 2.5.6.

Definition 4. *A label is extended using the extension function* $f(i,j) = l + [c_{ij}, r^1_{ij}, r^2_{ij}, \ldots r^{|R|}_{ij}] = l'$*, where* c_{ij} *is the cost of going from node* i *to* j *and* r^t_{ij}*,* $t \in R$*, is the resource consumption of going from node* i *to node* j*.*

The node j that is connected by the outgoing arcs δ^+_i from node i, is then used for adding new labels to the set of active labels using the extension function, which is defined in Definition 4. An extension is only valid if all of the resource constraints are satisfied, and also if the upper bound of the cost of the current node is less than or equal to the cost of the extended node.

2.5.3 Bellman-Ford algorithm

The Bellman-Ford algorithm [14] is used in the initialisation steps of MDDPA to calculate the shortest paths with respect to costs from a sink node to all other nodes in $G(V, A)$. The cost for the shortest path from a node $i \in V$ to the sink node is denoted by c^{sp}_i. Note that the resource constraints are not taken into consideration for this process, just the costs from going from one node to another.

The algorithm is shown in Algorithm 3, and the first step is the initialisation. An array with all costs for the nodes is set to infinity for all nodes $i \in V \setminus \{i_{sink}\}$, for the sink node the cost is set to 0. The next step is the relaxation of the edges. The relaxation step is used to compare the distance with other known distances for the nodes in order to continuously shorten the distances [15]. The relaxation step starts at the sink node, and compares the costs to all connecting nodes. If the cost of the sink node plus the cost of the connecting arc is less than the cost of the connecting node, the cost of the connecting node is set to the cost of the sink node in addition to the

Algorithm 3 Bellman-Ford algorithm

```
1: for each node i ∈ V \ {i_sink} do
2:     costs[i] = ∞
3: end for
4: costs[i_sink] = 0
5:
6: repeat |V| − 1 times
7:     for each arc (i, j) ∈ A with cost c do
8:         if costs[i] + c < costs[j] then
9:             costs[j] = costs[i] + c
10:        end if
11:    end for
```

cost of the connecting arc. For the first iteration the costs of all connecting nodes are all set to infinity, which means that the connecting nodes will be set to the cost of its connecting arc from the sink node. This process is then done for all nodes that are connected with an arc $(u, v) \in A$ and repeated $|V| - 1$ times. This will result in an array with the minimum costs of getting to node $i \in V$ from the sink node.

A common variant of the Bellman-Ford algorithm is to add detection for negative-weight cycles. A negative-weight cycle occurs when a network contains a cycle where the weights of the arcs sum to a negative number, thus meaning that there is not a shortest path since this cycle always will yield lower costs [16]. In our case the networks are acyclic, meaning that there is no need for checking if a network contains any negative-weight cycles.

2.5.4 Label storing procedure

Label storing procedure (LSP) is the process of filling the sets of stored labels for the neighbours of the subnetwork $\bar{G}(\bar{V}, \bar{A})$, where neighbours of a network are defined in Definition 5. The subnetwork used in LSP can be any connected subnetwork that is part of G, but when using LSP in a column generation context the subnetwork is chosen as the paths for the non-degenerate basic variables for the master problem [3]. In other words, the network is chosen as the variables in a basic feasible solution that are all strictly positive [17]. The algorithm for LSP is shown in Algorithm 4.

Definition 5. *For a subnetwork $\bar{G}(\bar{V}, \bar{A})$ of a network $G(V, A)$ with nodes V and arcs A, such that $\bar{V} \subset V$ and $\bar{A} \subset A$, a node $j \in V$ is said to be a neighbour of $\bar{G}$ if there is an arc $(i, j) \in A \setminus \bar{A}$ such that $i \in \bar{V}$.*

Algorithm 4 Label storing procedure (LSP)

```
1: S_i ← ∅, ∀i ∈ V
2: L_1 ← {[0, 0, ..., 0]}
3: L_i ← ∅, ∀i ∈ V \ {i_source}
4: for all i ∈ V̄ do
5:     Dominance(L_i)
6:     for all (i, j) ∈ δ_i^+ do
7:         T_j ← Extension(L_i, j)
8:         if (i, j) ∈ Ā then
9:             L_j ← L_j ∪ T_j
10:        else
11:            S_j ← S_j ∪ T_j
12:        end if
13:    end for
14: end for
15: return S_i ∀i ∈ V
```

The first steps of LSP is the initialisation, where the labels for the set of stored labels $\mathcal{S}_i \ \forall i \in V$ is set to the empty set. The active labels $\mathcal{L}_i$ are set to the empty set for $i \in V \setminus \{i_{source}\}$ and to a label with cost set to zero, and all resource consumptions set to zero for $i = i_{source}$. The algorithm then iterates through all nodes that belong to the subnetwork $\bar{G}$, and applies the dominance function described in Definition 2 [11]. Then for all arcs $(i, j) \in \delta_i^+$, where δ_i^+ are all outgoing arcs from a node i, a temporary set $\mathcal{T}_j$ is created using the extension function defined in Definition 4. The extension function is used to dynamically create new labels from existing ones [3]. If the arc (i, j) belongs to $\bar{A}$, then the temporary set of labels $\mathcal{T}_j$ is added to the active labels $\mathcal{L}_j$, otherwise the set of labels is added to the stored labels $\mathcal{S}_j$. The LSP function returns all sets of stored labels $\mathcal{S}_i \ \forall i \in V$.

Figure 2.1 shows the idea of LSP on an example network with only one vehicle, where the source node is denoted by s and the sink node is denoted by d. The subnetwork consists of the yellow nodes and the connecting arcs between them. LSP sets the labels from the nodes in the subnetwork as the active labels, meaning the labels in the yellow nodes are set to active labels. Then all labels in the nodes that have a connection from a yellow node are set as the stored labels. The labels in the rest of the nodes are not included

in the search.

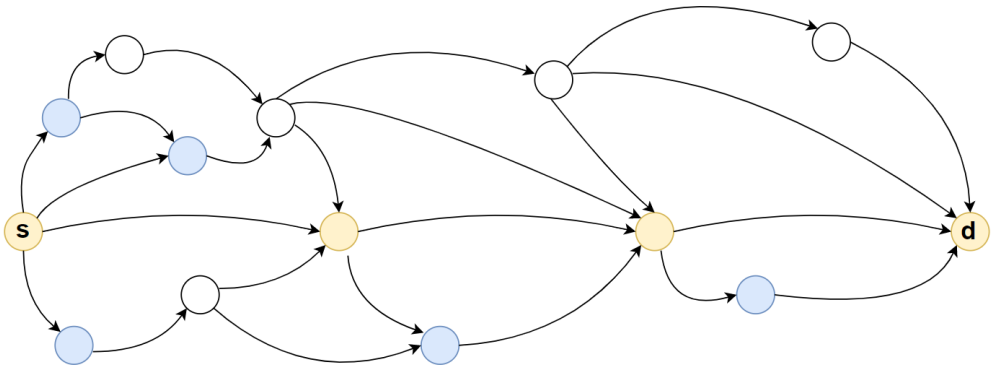

Figure 2.1: An example of a network with source node s and sink node d. The yellow nodes contain the active labels and the blue nodes contain the stored labels generated by LSP.

2.5.5 Label loading strategy

From LSP in Algorithm 4, a set of stored labels $\mathcal{S}_i$ at each node $i \in V$ is produced. These nodes can be explored with label loading strategies (LLS), and for this implementation a label loading strategy called nearest first (NF) is used. The idea behind LLS is for each node $i \in V$, there is a restricted search space induced by a subnetwork $G_i(V_i, A_i)$ of the network $G(V, A)$, where $V_i = \{j \in V, j \geq i\}$ and $A_i = \{(j, k) \in A, j \geq i\}$ [3].

2.5.5.1 Label loading strategy using nearest first

The strategy used for LLS is nearest first (NF), and is shown in Algorithm 5. The idea of NF is based on how close a node is to the sink node, where labels from a node closer to the sink node are prioritised. The set of labels in $\mathcal{S}$ are ordered topologically, and NF extends the label in reverse topological order. Algorithm 5 starts with initialising a length for the jumps, p_k, used to extend the labels. A jump is how many labels to select for each iteration k. For each iteration the sets of labels $\mathcal{S}_{i^*-1}, \mathcal{S}_{i^*-2}, \ldots, \mathcal{S}_j$ are selected in such a way that the equations in (2.10) are fulfilled, where i^* is the node from MDDPA in Algorithm 2. The length of the jumps is chosen as $p_k = p_0 \cdot r^k$, where $p_0 > 0$ is the first jump, and $r > 1$ is a ratio that determines how much the jumps increase. The node i is then chosen as $i = i^* - 1$ and then the steps 4–7 in Algorithm 5 are repeated until the length of the active labels

$|\mathcal{L}| \geq p_k$. These steps are used to add the stored labels $\mathcal{S}_i$ to the active labels $\mathcal{L}_i$.

$$\begin{cases} \sum_{i=j}^{i*+1} |\mathcal{S}_i| \geq p_k \\ \sum_{i=j+1}^{i*+1} |\mathcal{S}_i| < p_k \end{cases} \tag{2.10}$$

Algorithm 5 Label loading strategy (LLS), using nearest first (NF) strategy

1: $p_k \leftarrow p_0 \cdot r^k$
2: $i \leftarrow i^* - 1$
3: **while** $|\mathcal{L}| \leq p_k$ **do**
4: $\quad \mathcal{L}_i \leftarrow \mathcal{S}_i$
5: $\quad \mathcal{S}_i \leftarrow \emptyset$
6: $\quad \mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{L}_i$
7: $\quad i \leftarrow i - 1$
8: **end while**
9: $i^* \leftarrow i$

2.5.6 Learning from locally efficient labels

Learning from locally efficient labels (LLEL) is the procedure of using available information from previously generated labels in order to improve the solution process. LLEL is shown in Algorithm 6 [3].

LLEL compares all the labels from the set of active labels to the labels in the set of locally efficient labels. A locally efficient label is a label that is efficient at a given iteration, meaning that the label is feasible and is not dominated by any other label at the node it is currently at. In Algorithm 6, the label $l \in \mathcal{L}_i$ is removed if $l' \in \mathcal{P}_i$ dominates l, for node $i \in V$. On the other hand, if l dominates l' and if $l' \in \Pi_d$, where Π_i is the set of all locally efficient labels in $\mathcal{P}_i$ that has constructed a feasible path in the previous iteration, then the search for feasible descent directions begins, which is described in Section 2.5.6.1. Note that LLEL uses the function `CostBounding`, which is the function described in Definition 6 (see [3]).

Definition 6. *The function `CostBounding` is used to update the cost of the upper bound $\bar{c}_i$ at $i \in V$, whenever a feasible path π with better cost is found. The cost of the upper bound is given by $\bar{c}_i = c^{best} - c_i^{sp}$, where c^{best} is the cost of the best path found so far and c_i^{sp} is the cost from node $i \in V$ to the sink node calculated using the reverse shortest path.*

Algorithm 6 Learning from locally efficient labels (LLEL)

for all $l \in \mathcal{L}_i$ **do**
  for all $l' \in \mathcal{P}_i$ **do**
    if l' dominates l **then**
      $\mathcal{L}_i \leftarrow \mathcal{L}_i \setminus \{l\}$
    end if
    if l dominates l' **then**
      if $l' \in \Pi_i$ (FDD) **then**
        $l_i^\pi = l'$
        $\mathbf{d_i^\pi} = \mathbf{x}^\pi - \mathbf{x_i^\pi}$
        $\mathbf{x}^{\pi'} = \mathbf{x_i^l} + \mathbf{d_i^\pi}$
        $\Pi_d \leftarrow \Pi_d \cup \pi'$
        if $c_{\pi'} < c^{best}$ (IDD) **then**
          $c^{best} \leftarrow c_\pi$
          `CostBounding`(c^{best})
        end if
      end if
    end if
  end for
end for

2.5.6.1 Feasible descent directions

A descent direction is defined in Definition 7, and is a direction which leads closer to a local or global minimum [18]. For a descent direction to also be feasible, it means that the new point moved to using the descent direction is also feasible. In the context of LLEL, a descent direction is used to define potential directions that may lead to more efficient paths using the labels. For an iteration of MDDPA, let $\pi \in \Pi_i$ be a feasible path. The set of locally efficient labels that has contributed to constructing π consists of labels $l_i^\pi, i \in \mathcal{N}^\pi$, where $\mathcal{N}^\pi$ is the set of nodes that the path π passes through. Let $\mathbf{x}^\pi$ be the solution vector, where $\pi \in \Pi_i$, and let $\mathbf{x}_i^\pi$,where $\pi \in \Pi_i$ and $i \in V$, be the vector corresponding to the partial path l_i^π that terminates at node $i \in \mathcal{N}^\pi$. The vector $\mathbf{x_i^\pi}$ is given by

$$(\mathbf{x_i^\pi})_a = \begin{cases} 1, \text{if arc } a \text{ of the partial path is associated with } l_i^\pi, \\ 0, \text{otherwise.} \end{cases} \tag{2.11}$$

Definition 7. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a given function, and let $\mathbf{x} \in \mathbb{R}^n$ be a vector such that $f(\mathbf{x})$ is finite. A vector $\mathbf{p} \in \mathbb{R}^n$ is said to be a descent direction with respect to f at $\mathbf{x}$ if*

$$\exists \delta > 0 \text{ such that } f(\mathbf{x} + \alpha \mathbf{p}) < f(\mathbf{x}) \text{ for every } \alpha \in (0, \delta].$$

For LLEL, the descent direction is given by $\mathbf{d}_\mathbf{i}^\pi = \mathbf{x}^\pi - \mathbf{x}_\mathbf{i}^\pi$ and is a direction generated by a path $\pi \in \Pi_d$ at node $i \in V$. A complete path from source node to sink node is then defined by $\mathbf{x}^{\pi'} = \mathbf{x}_\mathbf{i}^\mathbf{l} + \mathbf{d}_\mathbf{i}^\pi$, where $\mathbf{x}_\mathbf{i}^\mathbf{l}$ is the vector for label l_i at node $i \in V$. The path π' might not be feasible and it is therefore necessary to check if the descent direction $\mathbf{d}_\mathbf{i}^\pi$ is a feasible descent direction, according to Definition 8. The descent direction $\mathbf{d}_\mathbf{i}^\pi$ could also be an improving descent direction (IDD), which is defined in Definition 9 [3].

Definition 8. *For a set of labels $\mathcal{L}_i$ at node $i \in V$, if there is a feasible path π of cost c_π and descent direction $\mathbf{d}_\mathbf{i}^\pi$, the descent direction $\mathbf{d}_\mathbf{i}^\pi$ is said to be a feasible descent direction (FDD) if there is a label $l_i \in \mathcal{L}_i$ such that the path π' given by $\mathbf{x}^{\pi'} = \mathbf{x}_\mathbf{i}^\mathbf{l} + \mathbf{d}_\mathbf{i}^\pi$ is feasible and the cost $c_{\pi'} \leq c_\pi$, where $c_{\pi'}$ is the cost for path π'.*

Definition 9. *An FDD is said to be an improving descent direction (IDD) if $c_{\pi'} \leq c^{best}$, where $c_{\pi'}$ is the cost of path π' and c^{best} is the cost of the best path found so far.*

Using the definitions 8 and 9, this can be reformulated using the context of MDDPA. If the label $l \in \mathcal{L}_i$ at node $i \in V$ is dominating label l_i^π, where $\pi \in \Pi_d$ is a feasible path, then $\mathbf{d}_\mathbf{i}^\pi$ is an FDD. This is because of for a label $l \in \mathcal{L}_i$ and a label $l_i^\pi \in \mathcal{P}_i$ where π is a feasible path with cost c_π, if the l_i dominates l_i^π, as in definition 2, the cost and all of the resource consumptions for l_i is less than or equal to the cost and resource consumptions for l_i^π. This means that the feasible path π' given by $\mathbf{x}^{\pi'} = \mathbf{x}_\mathbf{i}^\mathbf{l} + \mathbf{d}_\mathbf{i}^\pi$ has the cost $c_{\pi'} = c_l + c_{\mathbf{d}_\mathbf{i}^\pi} \leq c_{l^\pi} + c_{\mathbf{d}_\mathbf{i}^\pi} = c_\pi$, where $c_{\mathbf{d}_\mathbf{i}^\pi}$ is the cost of $\mathbf{d}_\mathbf{i}^\pi$. Since π' is a feasible path with a lower cost than the path π, it implies that the descent direction $\mathbf{d}_\mathbf{i}^\pi$ is an FDD. Using the same procedure, an FDD is said to be an IDD if $c_{\pi'} \leq c^{best}$.

2.6 Test data

A total number of 39 test instances are going to be used in order to test and evaluate the implementation of MDDPA. The test data consists of different types of problems. Things that could differ between the test instances are for instance whether there are cumulative rules or not, the size of the network, the number of vehicles, and the number of legs.

The test instances that are going to be used are instances that take up to one hour to run when using Jeppesen's algorithm. The currently used algorithm is implemented using multiple threads, which MDDPA does not do. In order to make a fair comparison, the test instances are going to be run on a single thread. All execution times for Jeppesen's algorithm can be found in Appendix A.1, note that the execution times are from running the instances on a single thread and the execution times are therefore slower than when running the algorithm on several threads.

The test instances used are described in Tables 2.1–2.5, where they are categorised by execution time, number of vehicles, number of legs, the ratio between the number of vehicles and the number of legs, and whether they have global constraints or not. All information about the tests can be found in Appendix B, including information about the cumulative rules. Note that for the execution times in Table 2.1, the instances are categorised by the execution time when running on multiple threads with Jeppesen's algorithm.

Table 2.1: Number of test instances categorised by running time and with/without cumulative rules for Jeppesen's current algorithm.

Execution time	$[0, 10s]$	$(10s, 1min]$	$(1min, 10min]$	$(10min, 1h]$
With cumulative rules	9	12	3	2
Without cumulative rules	1	9	3	0
Total	10	21	6	2

Table 2.2: Number of test instances categorised by number of vehicles and with/without cumulative rules.

Number of vehicles	$[0, 10]$	$(10, 20]$	$(20, 50]$	$(50, 279]$
With cumulative rules	13	3	9	1
Without cumulative rules	0	2	9	2
Total	13	5	18	3

Table 2.3: Number of test instances categorised by number of legs and with/without cumulative rules.

Number of legs	$[0, 2500]$	$(2500, 5000]$	$(5000, 10000]$	$(10000, 14948]$
With cumulative rules	6	0	13	7
Without cumulative rules	2	11	0	0
Total	8	11	13	7

Table 2.4: Number of test instances categorised by the ratio between the number of vehicles and the number of legs, and with/without cumulative rules.

Ratio	$(0, 250]$	$(250, 500]$	$(500, 1000]$	$(1000, 2990]$
With cumulative rules	14	2	2	8
Without cumulative rules	13	0	0	0
Total	27	2	2	8

Table 2.5: Number of test instances categorised by whether they have global constraints or not, and with/without cumulative rules.

Has global constraints?	Yes	No
With cumulative rules	4	22
Without cumulative rules	0	13
Total	4	35

3 | Method

This chapter consists of two sections. The first section describes how the implementation of the algorithm was carried out, and how the implementation differs from the theoretical description of the algorithm. The second section describes the testing and evaluation of the algorithm.

3.1 Implementation of MDDPA

Our implementation of the algorithm is very similar to how it was described by Himmich et al. [3], with a few minor changes due to how the current code at Jeppesen is implemented and a few changes in order to try to make the execution more efficient. The implementation was done in the programming language `C++`.

Starting from the beginning of MDDPA in Algorithm 2 at line 1, the subnetwork was defined as the non-degenerate basic variables for the master problem, as described in Section 2.5.5. These non-degenerate basic variables were generated by the previous iteration of the column generation. For the first iteration of the column generation, the subnetwork was set to all nodes in the network. In other words, the subnetwork for the first iteration of the column generation was set to the full network.

The implementation of the Bellman-Ford algorithm, used at line 2 in Algorithm 2 was implemented the same way as it was described in Algorithm 3, with the exception that the values were passed as a parameter to MDDPA instead of being calculated each iteration. This was done because the reverse shortest path only needs to be calculated once for each vehicle, since it is constant between the iterations for the column generations.

Continuing at lines 3–8, the implementation was the same as described in Algorithm 2. However, the value for c^{best} was set to the maximal value for a double in `C++`, about $1.8 \cdot 10^{308}$ [19]. The function for the label storing

procedure was implemented in a similar way as described in Algorithm 4, but with some added checks. For instance if there are no labels at a node i, the function continues to the next node instead of calling the extension function. This was done in order to try to improve the execution time of the algorithm by not calling functions when not needed. The rest of the initialisation steps in Algorithm 2 that differed in the implementation was the initial value for i^*. The nodes for our networks are ordered in such a way that all source nodes have the lowest indices, and the sink nodes have the largest indices. All of the test instances used had at least two vehicles, meaning that if setting $i^* = |V| - 1$ as in Algorithm 2, i^* could be set to the index of another sink node or a node that does not connect to the sink node. The initial value for i^* was instead set to the index of the last node connecting to the sink node for each vehicle.

The search procedure, lines 12–24 in Algorithm 2, was mainly implemented in the same way as described. Some things that differed in the implementation were the label loading strategy and the learning from locally efficient labels. In the label loading strategy, no labels were removed as described in Algorithm 5. Instead there is a check if the node has already been visited in a previous iteration in order to not insert the same labels to the active labels over and over again. The learning from locally efficient labels is implemented in a similar way as described in Algorithm 6, except no labels are removed from the active labels if they are dominated by a label in the locally efficient labels. For the whole algorithm, no labels are ever removed from a set if they are dominated. Instead there is a check in the extension function, if a newly created label is dominated by any other label in the set it is not added to the set.

3.2 Evaluation of the results

After the implementation was done, the code was compiled and tested on a number of different test instances. The instances were run on a single thread in order to make the comparisons as fair as possible since there is no multithreading implemented in MDDPA. The results from running the instances showed a number of different things, of interest were mainly the cost and the execution time. In this context, cost refers to the total sum of all the best feasible routes found for each vehicle in the network. The execution time is the total time for solving the problem, this includes all column generations and the time for solving the pricing problem, as well as everything that is done before and after the column generation. The cost and the execution time

are saved and a comparison between the values for MDDPA and Jeppesen's algorithm is made. Even though MDDPA and Jeppesen's algorithm are used in only the column generation iteration, MDDPA and Jeppesen's algorithm from now on refers to the full process when using each of the algorithms respectively.

4 | Results

This chapter presents the results. The results from the test instances are split into two parts, results for the instances that include cumulative rules and results for the instances that do not include cumulative rules.

Each test instance has a unique index, in order to easily compare the same test results between the two different algorithms. All information and execution times for the tests can be found in appendices A and B.

4.1 Test instances with cumulative rules

Table 4.1 shows how the implementation of MDDPA compared to the currently used algorithm, when running the tests on a single thread. The table shows the relative change between the cost and the execution time for MDDPA and Jeppesen's algorithm. The relative change in cost is calculated by

$$c_{change} = \frac{c_{MDDPA} - c_{Jeppesen}}{c_{Jeppesen}},$$

where c_{MDDPA} is the cost found by using MDDPA and $c_{Jeppesen}$ is the cost found by the currently used algorithm. The relative change in execution time is calculated in the same way as for the cost. For instance the test instance with index 1, the cost was 6.22% lower than the cost found with Jeppesen's algorithm. But it took MDDPA 37.29% longer time.

The average cost for all test instances is 3.73% higher with a standard deviation of 31.59% and a median cost that is the same as the costs for Jeppesen's algorithm. The average execution time for all test instances is 44.91% longer with a standard deviation of 58.35%, than the execution time for Jeppesen's algorithm, and a median execution time that is 34.65% longer than that of Jeppesen's algorithm.

Figures 4.1 and 4.2 visualise the difference in cost and execution time for MDDPA in relation to Jeppesen's algorithm.

Table 4.1: The cost and execution time when using MDDPA compared to Jeppesen's algorithm for tests with cumulative rules.

Instance index	**Relative change, cost [%]**	**Relative change, time[%]**
1	-6.22	+37.29
2	-6.22	+22.65
3	0	+20.93
4	0	+22.22
5	0	+62.50
6	0	-29.41
7	0	-15.79
8	0	+35.44
9	-0.01	+115.87
10	-0.02	+98.48
11	-0.01	+114.40
12	0	+108.91
13	0	+86.55
14	+0.51	+14.06
15	-0.54	-0.91
16	0	+50.00
17	+139.97	0
18	0	0
19	+59.98	-25.00
20	0	+200.00
21	-22.61	+34.46
22	-22.62	+40.11
23	-22.61	+34.83
24	-22.67	+156.59
25	0	+12.93
26	+0.02	-29.41
All tests, mean	+3.73	+44.91
All tests, median	0.00	+34.65

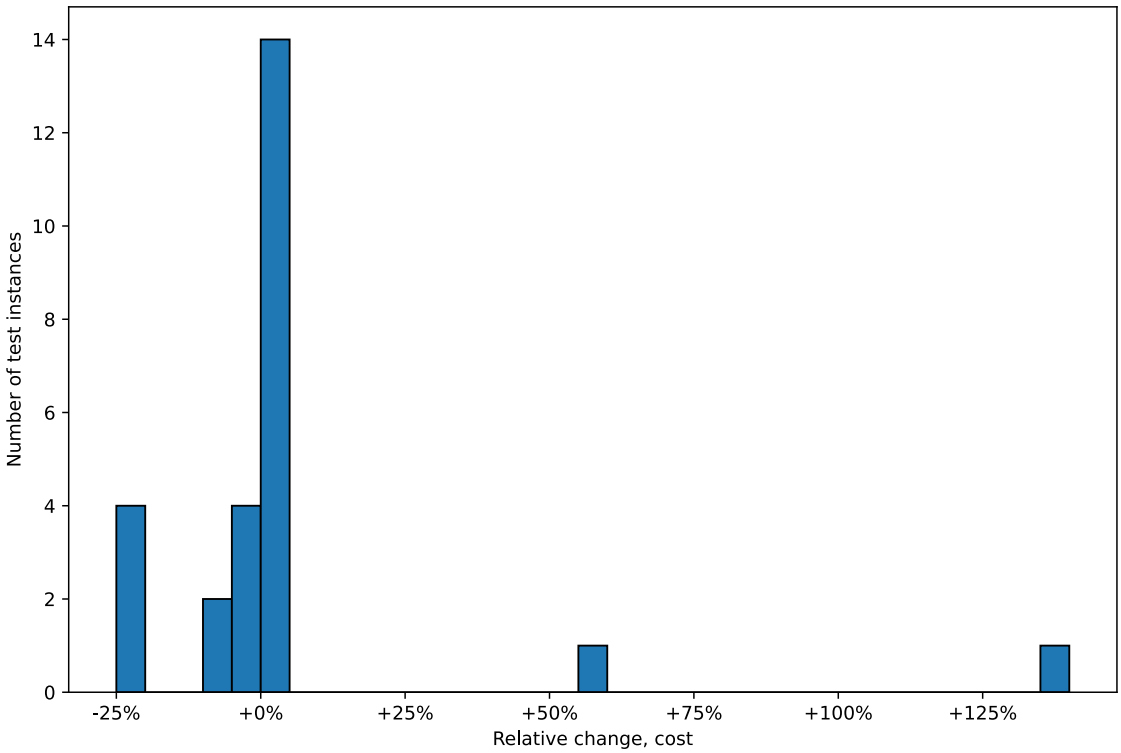

Figure 4.1: Histogram of relative change in cost for test instances with cumulative rules, for MDDPA compared to Jeppesen's algorithm.

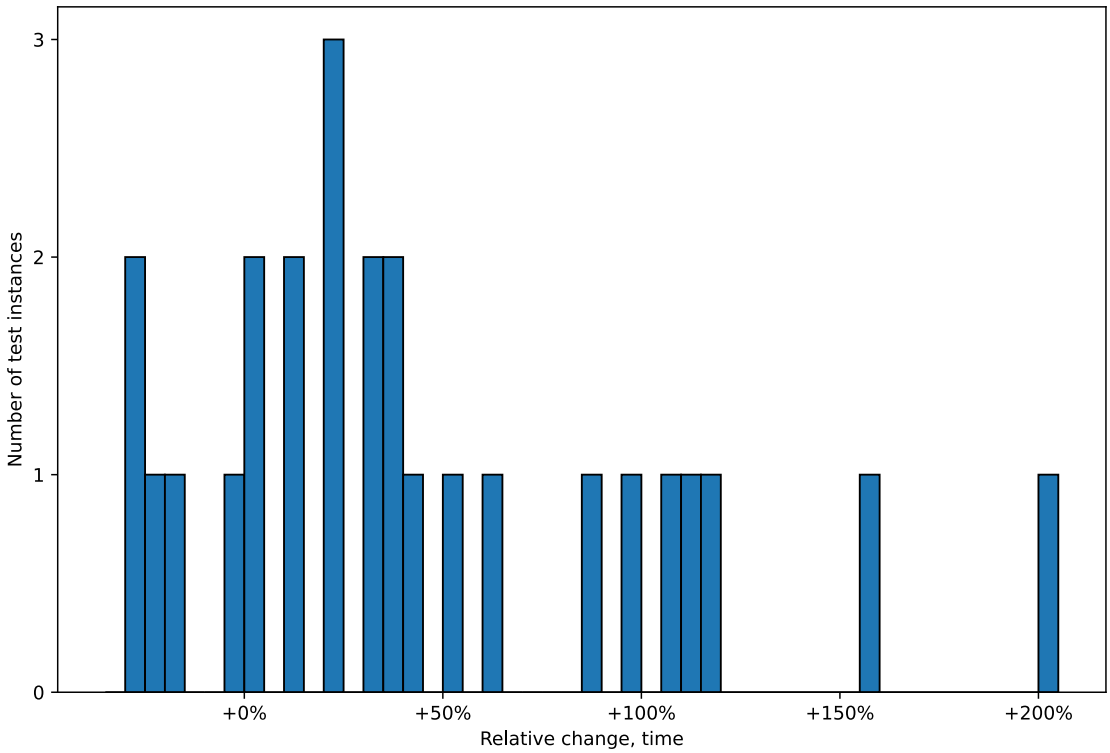

Figure 4.2: Histogram of relative change in execution time for test instances with cumulative rules, for MDDPA compared to Jeppesen's algorithm.

4.2 Test instances without cumulative rules

How MDDPA compares with the currently used algorithm for tests without cumulative rules are shown in table 4.2.

The average costs for all tests are 30.42% worse — with a standard deviation of 111.02% — than Jeppesen's algorithm, but with a median cost that is 0.01% better than the costs from Jeppesen's algorithm. The average execution time is 89.09% worse with a standard deviation of 217.26% than the currently used algorithm, and a median execution time that is 52.04% worse than the currently used algorithm.

Figures 4.3 and 4.4 visualise the difference in cost and execution time for MDDPA in relation to Jeppesen's algorithm.

Table 4.2: The cost and execution time when using MDDPA compared to Jeppesen's algorithm for tests without cumulative rules.

Instance index	**Relative change, cost [%]**	**Relative change, time [%]**
27	-0.01	+50.85
28	+399.91	+62.61
29	-0.02	+58.06
30	-0.02	+52.04
31	-0.02	+64.04
32	-0.03	+70.00
33	-0.01	+55.56
34	-0.02	+40.15
35	0	-33.33
36	0	+800.00
37	-4.32	-45.77
38	-0.01	-8.81
39	+0.01	-7.19
All tests, mean	+30.42	+89.09
All tests, median	-0.01	+52.04

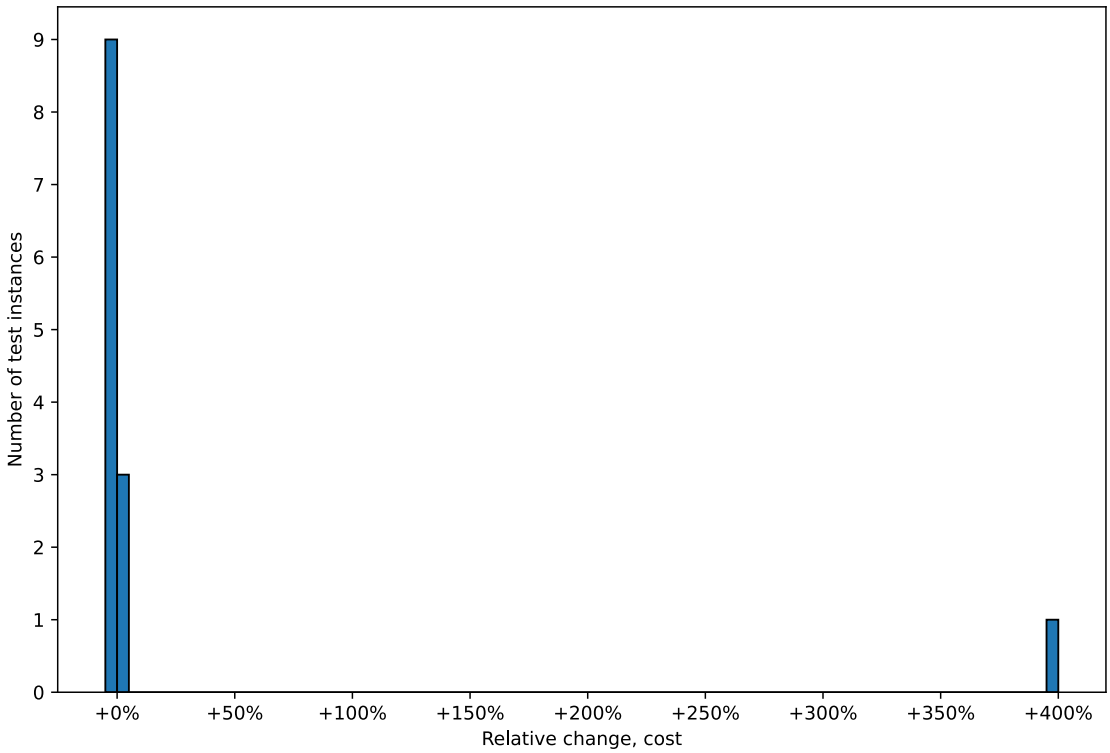

Figure 4.3: Histogram of relative change in cost for test instances without cumulative rules, for MDDPA compared to Jeppesen's algorithm.

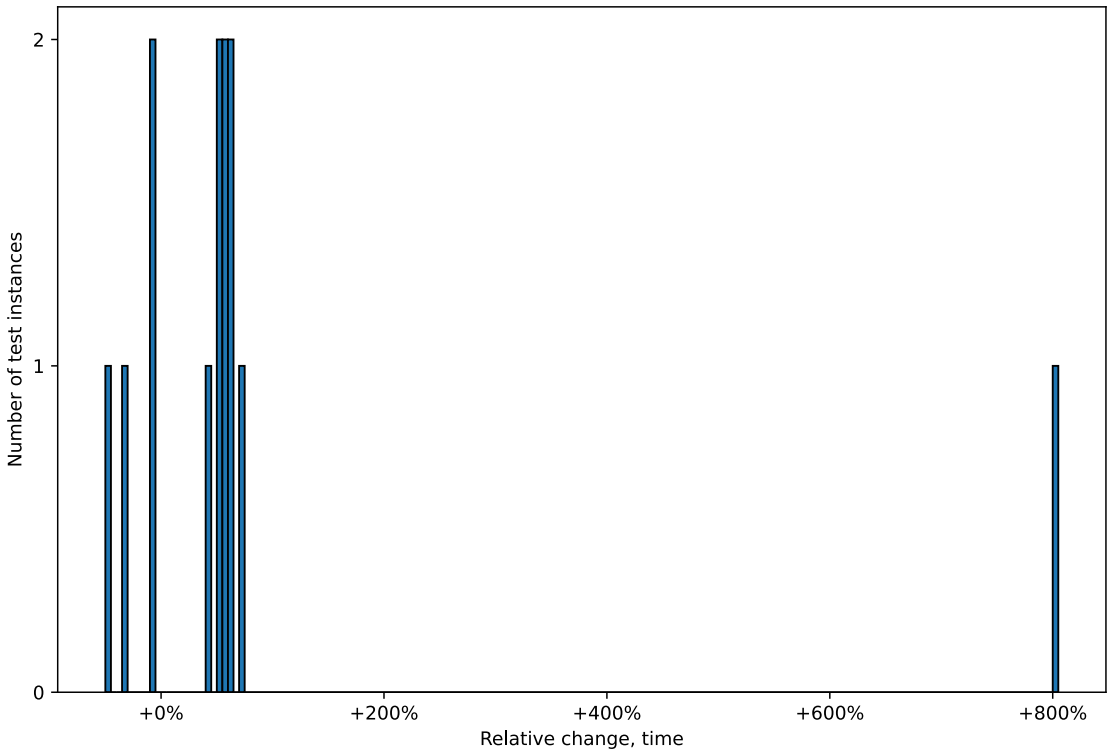

Figure 4.4: Histogram of relative change in execution time for test instances without cumulative rules, for MDDPA compared to Jeppesen's algorithm.

5 Discussion

This chapter discusses the results from the previous chapter and compares the results from using MDDPA to using the currently used algorithm at Jeppesen. There are four different sections, two sections that discuss the comparisons between the costs and execution times separately, and one section that discuss the overall performance with the costs and execution times combined. The last part discusses some suggestions for further research and improvements.

5.1 Comparison between costs

Looking at the relative change in cost in Tables 4.1 and 4.2, together with Figures 4.1 and 4.3, the cost found by MDDPA was for 33 out of 39 test cases at least as good, or better, than the cost found by Jeppesen's algorithm. The results for MDDPA varies by quite a lot however, with some cost found that were as high as four times the cost found by Jeppesen's algorithm.

Comparing the results between the tests with and without cumulative rules, the average relative change in cost for the tests with cumulative rules is lower than the average relative change in cost without cumulative rules, the median relative change in cost is however very similar for the two groups. On the other hand, MDDPA found in six cases a cost that was at least 5% better than the cost found by Jeppesen's algorithm for tests with cumulative rules. For tests without cumulative rules, MDDPA did not find any costs that were over 5% better. The lowest cost found by MDDPA was however often very similar to the cost found by the currently used algorithm. For tests with cumulative rules, the best cost found was within a relative change of 1% between the two algorithms for 18 out of 26 tests (69.2%). For tests without cumulative rules, the amount of tests that had a relative change within 1% were 11 out of 13 (84.6%). Only two tests with cumulative rules, and one test without cumulative rules, had a relative change in cost that was over 1%

worse. This might indicate that there is not a big difference between tests with and without cumulative rules in regards to MDDPA finding a cost that is at least as good as the cost found by the currently used algorithm. However, MDDPA seems to be better at finding better solutions for test instances with cumulative rules compared to the instances without cumulative rules.

Test instances 21–24 yielded the best results with regard to cost. The best cost found by MDDPA was at least 22% lower than the cost found by the currently used algorithm. Looking at the descriptions of the instances in Table B.4 in Appendix B, these instances were the only cases that had global constraints. Since the cost for MDDPA was this much better, it might indicate that MDDPA is better at handling global constraints than Jeppesen's algorithm.

The test instances that MDDPA performed the worst at, with regards to cost, were instances 17, 19, and 28. Looking at Tables B.4 and B.6 in Appendix B, the things that mainly stand out are that tests 17 and 19 both have very few legs, only 12 each, while the average number of legs for all tests is approximately 6300. These instances also contain a very small amount of nodes. MDDPA uses a subnetwork as described in Section 2.5.4 in order to search for improved solutions. If the network is very small, there is a high risk that the resulting subnetwork is going to be set as the full network and thus making it harder for the algorithm to search for improving feasible routes.

The test instances with cumulative rules can be categorised into six different categories in order to compare them. The different categories are shown in Table 5.1. Which category each instance belongs to is shown in Table **??**, note that some tests are in multiple categories since they have multiple cumulative rules. For each category, the mean value together with the standard deviation, and also the median value of the cost are shown in Table 5.2.

Table 5.1: The different categories for the cumulative rules.

Category	Description	Test instances
A_{short}	Stops at a suitable airport every x days, where $x \leq 3$	21, 22, 23, 24, 25, 26
A_{long}	Stops at a suitable airport every x days, where $x > 3$	26
B_{short}	Stops at a home or maintenance base every x days, where $x \leq 3$	1, 2, 9, 10, 11, 12, 13
B_{long}	Stops at a home or maintenance base every x days, where $x > 3$	3, 4, 5, 6, 14, 15, 25
L	Cumulative constraints regarding flight legs	7, 8
O	Other cumulative rules	16, 17, 18, 19, 20

Table 5.2: Mean and median values for the cost found for the test instances. The standard deviation are shown inside the parentheses.

Category	Mean (Standard deviation) [%]	Median [%]
A_{short}	-15.08 (10.67)	-22.61
A_{long}	+0.02 (0)	+0.02
B_{short}	-1.78 (2.80)	-0.01
B_{long}	0.00 (0.28)	0
L	0.00 (0)	0
O	+39.99 (55.12)	0

It is clear that the categories with shorter stop intervals (A_{short} and B_{short}) result in lower costs compared to the other categories, but the sample size for all categories is very small, meaning it is hard to draw any conclusions. The results for the test instances with shorter stop intervals are also more scattered, meaning that they have a higher standard deviation, compared to the categories with longer intervals. The instances that performed the worst were the tests in category O. These are instances that did not fit into any of the other categories. The mean cost for this category were much higher than the rest of the categories, but the results were also a lot more scattered.

5.2 Comparison between execution time

The results for the execution times, as seen in Tables 4.1 and 4.2 together with Figures 4.2 and 4.4, also varies by quite a lot. Some test instances have much lower execution times, while some have much longer execution times. Compared to the relative change in cost, the relative change in execution time is instead usually higher. For 28 out of 39 test cases, the execution times were longer when using MDDPA. Comparing the results for the test instances with and without cumulative rules, MDDPA was just as fast or faster than Jeppesen's algorithm for about 26.9% of the instances with cumulative rules and about 30.8% of the instances without cumulative rules. Overall MDDPA seems to be running faster for instances with cumulative rules, with both the mean and median relative change lower than for the instances without cumulative rules. However, the solutions with the largest negative relative change for execution time were for the test instances without cumulative rules, with solutions found where the execution time was up to approximately 50% lower when running MDDPA compared to the currently used algorithm. For instances without cumulative rules, the relative change in execution times are also more concentrated from +40% to +70%, where as for the instances with cumulative rules the relative change in execution times are more spread out.

The test instances that performed the best with regards to relative change in execution time were instances 26, 35, and 37. Instance 19 was also computed fast, but the cost found was almost 60% worse, which makes the comparison not very interesting. It is hard to tell why these in particular performed the best, the only main thing they have in common is that the ratio between the number of vehicles and number of legs are very similar, around 25-40. Test instance 15 also had a ratio close to 25-40, and had an execution time that was lower when using MDDPA, which might be an indication that MDDPA works well on these types of instances.

On the other hand, the test instances that performed the worst with respect to execution time were instances 20, 24, and especially 36. Instance 36 was a instance that had 199 vehicles and only one leg, and the results from this might not be very interesting since it might have been easier to schedule this manually. Test instance 20 was very similar to instance 17 and 19, which were discussed in the previous section where it seemed that MDDPA did not work as well as Jeppesen's algorithm for very small problems. Test instance 24 on the other hand was not a smaller problem, it was however one of the instances previously discussed that had global constraints. This instance

performs worse than the other instances with global constraints, and global constraints do not seem to be a factor in the execution time. It is not clear why this instance performs much worse than the rest.

Also in regards to total execution time, the tests with cumulative rules can be categorised in the same way as in Table 5.1. The mean and standard deviation for the execution time, together with the median value are shown in Table 5.3. The categories with longer intervals (A_{long} and B_{long}) performed better in general than the categories with shorter intervals, the total opposite of when looking at the costs. It should however be noted that there is only one instance in category A_{long} and the shorter execution time might depend on other things than which type of cumulative rule the test has. The test in category L also has lower execution times compared to many other categories, but there is also very few tests for this category. Category O did also not perform very well in regards to execution time.

Table 5.3: Mean and median values for the total running times for the test instances. The standard deviation are shown inside the parentheses.

Category	**Mean (Standard deviation)**	**Median**
A_{short}	+41.59% (56.55%)	+34.65%
A_{long}	-29.41% (0.00%)	-29.41%
B_{short}	+83.45% (35.29%)	+98.48%
B_{long}	+14.62% (25.56%)	+14.06%
L	+9.83% (25.62%)	+9.83%
O	+45.00% (81.24%)	0.00%

5.3 Overall comparison

Comparing the results overall it seems that MDDPA is good at finding routes with just as good or lower cost than the currently used algorithm, but the trade-off that the execution time usually is longer. For some situations this trade-off might be good, for instance finding a solution that is over 20% better, but with an execution time that is $30\% - 160\%$ longer. But for many test instances the reduction of cost is around $0.01\% - 0.02\%$, and having execution times that are over 50% longer might not be worth it.

Looking at Tables 5.2 and 5.3, and comparing the tests with cumulative rules, it might be advantageous to use MDDPA over Jeppesen's algorithm for some categories. For instance in category A_{short} it might be worth having

on average 41.59% longer execution times in return for on average 15.08% better costs. But for category B_{short} it is probably not worth having on average 83.45% longer execution times in return to just a small improvement in cost for using MDDPA.

5.4 Further improvements

There are two main things that could be improved and analysed further in order to improve the performance of MDDPA. The first thing is when running the column generation, when solving the SPPRC each vehicle gets solved sequentially, meaning the SPPRC gets solved for one vehicle at a time. In order to improve the execution time, the SPPRC could get solved in parallel for the vehicles in the network. In other words, using multiple threads in order to compute the SPPRC for the different vehicles at the same time. This will mainly benefit the problems with a large number of vehicles.

The second thing that could be interesting to analyse further is how the choice of parameters in the label loading strategy affects the results of the algorithm. In the label loading strategy a parameter p_k is used in order to decide how many labels to use in each iteration. The parameter p_k is set to $p_0 \cdot r^k$, where p_0 is the starting value, r is a ratio, and k is the iteration index. When solving the tests, p_0 was set to $N + 1$, where N is the number of nodes for the network used, and r was set to 2. A further analysis could be to investigate how much these parameters affect the result, and if there are optimal values for these parameters and what they would be in that case.

6 | Conclusions

Returning to the beginning of this study, the aim was to compare the costs and execution times by MDDPA and compare them to Jeppesen's algorithm. The results were very scattered, where some test instances yielded better results than others. Some results were very good, while others were not as good. All in all, MDDPA found for most instances a cost that was at least as good, or better, than the cost found by the currently used algorithm. On the other hand MDDPA usually had longer execution times. Finding a solution with lower cost usually came with the drawback of having a longer execution time, which in many cases might not be desirable.

The aim was also to investigate if MDDPA yielded better results for some categories of problems. Even though there was evidence that suggested that instances with global constraints gave better results with regards to cost and instances that had a ratio between the number of vehicles and number of legs in the interval around 25–40 gave better results with regards to execution time, it is hard to draw any conclusions whether how much these factors affected the results, since the results were so scattered and the sample size was very small. There might be even more factors that affect the results than those listed in the previous chapter.

Furthermore, there are several improvements to the implementation that could be done, for instance implementing parallelisation or testing how changing different parameters could have an effect on the result.

References

[1] Elena Mazareanu. *Global air traffic - number of flights 2004–2022.* Accessed: 2022-02-16. 2021.
URL: https://www.statista.com/statistics/564769/airline-industry-number-of-flights/.

[2] Mattias Grönkvist. "The Tail Assignment Problem".
PhD thesis. Chalmers University of Technology, 2005.

[3] Ilyas Himmich, Issmail El Hallaoui, and Francois Soumis.
A multidirectional dynamic programming algorithm for the shortest path problem with resource constraints. Tech. rep.
Montréal, Canada: Les Cahiers du GERAD, Feb. 2018.

[4] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin.
Network Flows. New Jersey, USA: Prentice-Hall, 1993, pp. 80–81.

[5] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs.*
London, England: Springer, 2009, p. 136.
DOI: 10.1007/978-1-84800-998-1.

[6] George B. Dantzig and Philip Wolfe.
"The Decomposition Algorithm for Linear Programs".
In: *The Econometric Society* (1961), pp. 767–778.

[7] George B. Dantzig and Mukund N. Thapa. *Linear Programming.*
New York, USA: Springer, 2003, p. 281.

[8] Edvin Åblad. *Dantzig–Wolfe decomposition.* Accessed 2022-03-10. 2019. URL: https://chalmers.instructure.com/courses/7420/files/276244/download?verifier=Pk1DynY0jHdzbBRt8V3W7sQi9yC2RvXygTWmMbMo&wrap=1.

[9] Lucas Létocart, Nora Touati Moungla, and Anass Nagih.
"Dantzig-Wolfe and Lagrangian decompositions in integer linear programming". In: *International Journal of Mathematics in Operational Research* (2011), p. 6.

[10] Michelle Hribar, Valerie E. Taylor, and David E. Boyce.
Choosing a Shortest Path Algorithm. Tech. rep.

Chicago, Illinois: Northwestern University and University of Illinois at Chicago, Sept. 1995.

[11] Martin Desrochers and Francois Soumis. "A Generalized Permanent Labelling Algorithm For The Shortest Path Problem With Time Windows". In: *INFOR: Information Systems and Operational Research* (1988), pp. 192–199.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* Massachusetts, USA: MIT Press, 2009, p. 359.

[13] Giovanni Righini and Matteo Salani. "Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints". In: *Discrete Optimization* (2006), pp. 260–266.

[14] Richard Bellman. "On a routing problem". In: *Quarterly of Applied Mathematics 16* (1958), pp. 87–90.

[15] Alex Chumbley, Karleigh Moore, Eli Ross, and Jimin Khim. *Bellman-Ford Algorithm.* Accessed: 2022-02-18. 2021. URL: https://brilliant.org/wiki/bellman-ford-algorithm/.

[16] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs.* London, England: Springer, 2009, p. 88. DOI: 10.1007/978-1-84800-998-1.

[17] John E. Mitchell. *Math Models of OR: Degeneracy.* Accessed: 2022-02-22. 2018. URL: http://eaton.math.rpi.edu/faculty/Mitchell/courses/matp4700/notesMATP4700/lecture05/05A%5C_degeneracybeamer.pdf.

[18] Niclas Andréasson, Anton Evgrafov, Michael Patriksson, Emil Gustavsson, Zuzana Nedělková, Kin Cheong Sou, and Magnus Önnheim. *An Introduction to Continuous Optimization.* Lund, Sweden: Studentlitteratur, 2018, p. 92.

[19] Microsoft. *Double.MaxValue Field.* Accessed 2022-05-03. 2022. URL: https://docs.microsoft.com/en-us/dotnet/api/system.double.maxvalue?view=net-6.0.

A | Results for the algorithms

The cost, execution time, number of vehicles, and number of legs are shown for the each test instance used in the following tables. Each instance were ran three times in order to calculate the execution time in case it differed by a lot between the runs. The values in the column for the execution time therefore shows a mean together with a standard deviation for the three runs. Each instance has a unique test number in order to easily compare the same test between the two different algorithms.

A.1 Currently used algorithm at Jeppesen Systems

A.1.1 Test instances with cumulative rules

Table A.1: Instances with cumulative rules with index 1-26, run with a single thread.

Instance index	Cost	Execution time (Mean) [hh:mm:ss]	Execution time (Standard deviation) [hh:mm:ss]
1	3216354400	00:00:59	00:00:01
2	3216315300	00:01:00	00:00:02
3	52913700	00:01:12	00:00:02
4	6851800	00:00:03	00:00:01
5	6851800	00:00:03	00:00:01
6	6851800	00:00:06	00:00:02
7	125216067	00:00:38	00:00:02
8	125213505	00:00:26	00:00:01
9	54514	00:01:24	00:00:00
10	54518	00:01:28	00:00:01
11	54510	00:01:23	00:00:01
12	54508	00:01:22	00:00:01
13	54510	00:01:54	00:00:00
14	83804130	00:15:53	00:00:06
15	1184465000	01:44:49	00:00:08
16	768000	00:00:01	00:00:01
17	320042	00:00:01	00:00:01
18	78	00:00:01	00:00:01
19	320042	00:00:01	00:00:01
20	128055	00:00:00	00:00:01
21	1729488	00:00:59	00:00:00
22	1729488	00:00:59	00:00:00
23	1729488	00:00:59	00:00:01
24	1730672	00:00:43	00:00:00
25	53878680	00:35:33	00:00:14
26	904153	00:00:06	00:00:01

A.1.2 Test instances without cumulative rules

Table A.2: Instances without cumulative rules with index 27-39, run with a single thread.

Instance index	Cost	Execution time (Mean) [hh:mm:ss]	Execution time (Standard deviation) [hh:mm:ss]
27	8934632	00:00:39	00:00:01
28	8934539	00:00:38	00:00:02
29	8934759	00:00:31	00:00:01
30	8934800	00:00:33	00:00:01
31	8934689	00:00:30	00:00:01
32	4091130	00:00:33	00:00:01
33	6369006	00:00:21	00:00:00
34	8934800	00:00:44	00:00:00
35	3619137	00:00:17	00:00:00
36	0	00:00:00	00:00:01
37	2545420	00:11:22	00:00:03
38	110684	00:12:36	00:00:03
39	110664	00:12:26	00:00:01

A.2 MDDPA

A.2.1 Test instances with cumulative rules

Table A.3: Instances with cumulative rules with index 1-26, run with a single thread.

Instance index	Cost	Execution time (Mean) [hh:mm:ss]	Execution time (Standard deviation) [hh:mm:ss]
1	3016370400	00:01:21	00:00:12
2	3016370400	00:01:14	00:00:00
3	52913000	00:01:27	00:00:03
4	6851800	00:00:04	00:00:01
5	6851800	00:00:04	00:00:01
6	6851800	00:00:04	00:00:00
7	125215426	00:00:32	00:00:15
8	125212700	00:00:36	00:00:01
9	54506	00:03:01	00:00:12
10	54506	00:02:54	00:00:06
11	54506	00:02:59	00:00:08
12	54506	00:02:52	00:00:03
13	54508	00:03:33	00:00:06
14	84234100	00:18:07	00:02:10
15	1178019000	01:43:52	00:02:05
16	768000	00:00:01	00:00:00
17	768000	00:00:01	00:00:01
18	78	00:00:01	00:00:01
19	512010	00:00:01	00:00:00
20	128055	00:00:01	00:00:00
21	1338380	00:01:19	00:00:10
22	1338288	00:01:23	00:00:07
23	1338459	00:01:20	00:00:09
24	1338364	00:01:50	00:00:13
25	53881310	00:40:08	00:03:13
26	904367	00:00:04	00:00:01

A.2.2 Test instances without cumulative rules

Table A.4: Instances without cumulative rules with index 27-39, run with a single thread.

Instance index	Cost	Execution time (Mean) [hh:mm:ss]	Execution time (Standard deviation) [hh:mm:ss]
27	8933336	00:00:59	00:00:00
28	44664359	00:01:02	00:00:00
29	8933337	00:00:49	00:00:00
30	8933337	00:00:50	00:00:00
31	8933337	00:00:49	00:00:00
32	4089821	00:00:57	00:00:00
33	6368652	00:00:33	00:00:00
34	8933337	00:01:02	00:00:00
35	3619137	00:00:11	00:00:00
36	0	00:00:03	00:00:00
37	2435340	00:06:10	00:00:00
38	110678	00:11:30	00:00:00
39	110678	00:11:32	00:00:00

B | Information about the test instances

Information about the test instances are listed in the following tables. Information regarding the number of vehicles and number of legs are listed together with a ratio between the number of vehicles and number of legs. The higher the ratio is the larger the number of legs are compared to the number of vehicles. Information about the methods used in the column generation are also listed. Table B.1 is a list over the acronyms used in tables B.4 and B.6. Table B.5 lists the number of cumulative rules for the tests and the name of the cumulative rules. Tables B.2 and B.3 lists what the names mean.

Table B.1: Acronyms used for Tables B.4 and B.6.

Acronym	**Meaning**
CF	Constraint Fixing
HF	Hybrid Fixing
PS	Primal Simplex
S	Subgradient

Table B.2: The meaning of the different cumulative rules, part 1.

Name of cumulative rule	**Meaning**
Z rule	Visit maintenance base during night time at least once every six days
Special Z rule (2)	Visit maintenance base during night time at least once every two days
Special Z rule (4)	Visit maintenance base during night time at least once every four days
A check	Visit maintenance base during night time at least once every 450 flying hours
100h check	Visit maintenance base during night time at least once every 100 flying hours
pen_below_min_legs	Soft rule. Count number of flight legs, never reset, and penalise quadratically if less than 15 flight legs assigned
Short maintenance base visit every X calendar days	Visit maintenance base for at least 4 hours at least every three days

Table B.3: The meaning of the different cumulative rules, part 2.

Name of cumulative rule	**Meaning**
Base visit rule	Visit maintenance base during night time at least once every five days
Maintenance Base visit rule	Soft rule, quadratic penalty above limit. Count flight hours, reset at maintenance base visits. Limit 4 or 6 days depending on aircraft type
maintenance.maintenance_rule_1_label_p	This rule can be customised to limit different things, with different limits
B-73W Maintenance Base visit rule (Days)	Visit maintenance base at least once every six days
B-73W Home Base visit rule (Days)	Visit home base at least once every six days
735: Max 1 midnight between 2 hour maintenance	Max 1 midnight between 2 hour maintenance, only applies to 735 aircraft
Periodic mandatory stop rule	Six hours stop at any airport at least every 48 hours
Long maintenance layover rule	At least eight hours stop at a suitable airport every 7 days
Short maintenance layover rule	At least four hours stop at a suitable airport every 48 hours

B.1 Test instances with cumulative rules

Table B.4: Information about test instances with cumulative rules.

Instance index	IP Method	LP Method	Global constraints	Vehicles	Legs	Ratio
1	CF	S	No	17	10044	590.8235
2	CF	S	No	17	10044	590.8235
3	CF	S	No	9	14948	1660.889
4	CF	S	No	4	11959	2989.75
5	CF	S	No	4	11959	2989.75
6	CF	S	No	4	11959	2989.75
7	CF	S	No	30	8188	272.9333
8	CF	S	No	30	8188	272.9333
9	CF	S	No	39	6111	156.6923
10	CF	S	No	39	6111	156.6923
11	CF	S	No	39	6111	156.6923
12	CF	S	No	39	6111	156.6923
13	CF	S	No	39	6111	156.6923
14	CF	S	No	43	6534	151.9535
15	CF	S	No	279	12260	43.94265
16	CF	S	No	2	12	6
17	CF	S	No	2	12	6
18	CF	S	No	4	12	3
19	CF	S	No	6	12	2
20	CF	S	No	8	12	1.5
21	CF	S	Yes	4	5380	1345
22	CF	S	Yes	4	5380	1345
23	CF	S	Yes	4	5380	1345
24	CF	S	Yes	4	5380	1345
25	CF	S	Yes	30	5576	185.8667
26	CF	S	No	15	560	37.33333

Table B.5: Number of cumulative rules and type of cumulative rules for the test instances.

Index	Number of cumulative rules	Type of cumulative rule
1	1	Special Z rule (2)
2	1	Special Z rule (2)
3	3	A check Z rule Special Z rule (4)
4	3	Z rule A check 100h check
5	3	Z rule A check 100h check
6	3	Z rule A check 100h check
7	1	pen_below_min_legs
8	1	pen_below_min_legs
9	1	Short maintenance base visit every X calendar days
10	1	Short maintenance base visit every X calendar days
11	1	Short maintenance base visit every X calendar days
12	1	Short maintenance base visit every X calendar days
13	1	Short maintenance base visit every X calendar days
14	2	B-73W Maintenance Base visit rule (Days) B-73W Home Base visit rule (Days)
15	1	Base visit rule
16	1	maintenance.maintenance_rule_1_label_p
17	1	maintenance.maintenance_rule_1_label_p
18	1	maintenance.maintenance_rule_1_label_p
19	1	maintenance.maintenance_rule_1_label_p
20	1	maintenance.maintenance_rule_1_label_p
21	1	735: Max 1 midnight between 2 hour maintenance
22	1	735: Max 1 midnight between 2 hour maintenance
23	1	735: Max 1 midnight between 2 hour maintenance
24	1	735: Max 1 midnight between 2 hour maintenance
25	2	Periodic mandatory stop rule Maintenance Base visit rule
26	2	Long maintenance layover rule Short maintenance layover rule

B.2 Test instances without cumulative rules

Table B.6: Information about tests without cumulative rules.

Instance index	IP Method	LP Method	Global constraints	Vehicles	Legs	Ratio
27	CF	PS, S	No	40	4381	109.525
28	CF	PS, S	No	40	4381	109.525
29	CF	S	No	40	4381	109.525
30	CF	S	No	40	4381	109.525
31	CF	S	No	40	4381	109.525
32	CF, HF	PS, S	No	40	4381	109.525
33	CF	S	No	24	2987	124.4583
34	CF	S	No	40	4381	109.525
35	CF	S	No	22	621	28.22727
36	CF	S	No	199	1	0.005025
37	CF	S	No	71	2532	35.66197
38	CF	S	No	13	2906	223.5385
39	CF	S	No	13	2906	223.5385

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY