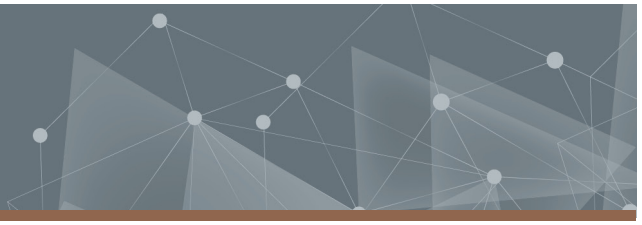
CHALMERS
UNIVERSITY OF TECHNOLOGY



Sequentially connected 2D assignment problems solved using Lagrangian dual methods

Application to the evaluation of multi-object tracking algorithms

Master's thesis in Engineering Mathematics and Computational Science

CAMILLA CARLSSON
HANNA EKELUND

DEPARTMENT OF MATHEMATICAL SCIENCES

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

Master's thesis 2022

Sequentially connected 2D assignment problems solved using Lagrangian dual methods

Application to the evaluation of multi-object tracking algorithms

CAMILLA CARLSSON
HANNA EKELUND

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Sequentially connected 2D assignment problems solved using Lagrangian dual methods
Application to the evaluation of multi-object tracking algorithms
CAMILLA CARLSSON
HANNA EKELUND

Supervisor: Ann-Brith Strömberg, Department of Mathematical Sciences
Examiner: Ann-Brith Strömberg, Department of Mathematical Sciences

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

Sequentially connected 2D assignment problems solved using Lagrangian dual methods
Application to the evaluation of multi-object tracking algorithms
CAMILLA CARLSSON, HANNA EKELUND
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

Multi-object tracking (MOT) algorithms are used to track different objects over time. The GOSPA metric has been used to accurately quantify object tracking in one time step, but it does not take into account the so-called track switches. This thesis will explore how an extended GOSPA metric can be solved efficiently. The extended GOSPA metric is modeled as a mixed binary linear program, which consists of several connected 2D assignment problems. To solve these, we utilize LP relaxation which results in a lower bound on the optimal value unless the problem possesses integrality property, in which case it equals the optimal value. We have implemented two methods to quantify MOT algorithms, multi-block ADMM, and a subgradient method. Our results show that both algorithms work well and scale linearly with the number of time steps and cubic with the number of estimates and ground truths. Thanks to a well-working heuristic which can be used as initialization, both methods converge fast for most instances. The heuristic finds optimal values for all besides one of our test instances. The optimal solutions to all instances are integer, indicating that the. Because of this, we formulate a conjecture that the connected assignment problem also possesses integrality property, although the constraint matrix is not totally unimodular. To improve the results, we suggest that one should prove or disprove the conjecture, to know whether the problem always has integrality property or not. Also, one should improve the numerical solution of the 2D assignment problems, since most of the computing time is spent on the solution of the subproblems.

Keywords: Multi-object tracking, Connected assignment problems, Subgradient, ADMM, Integrality property, Mixed binary linear program, LP relaxation

Acknowledgements

First of all, we would like to thank our supervisor and examiner Ann-Brith Strömberg for all your support and ideas during our thesis. Without your smart tips and tricks, the subgradient implementation would not have become as good. Thank you for all your courses in mathematical optimization that have made our interest in the subject to grow.

Next, we would like to thank Lennart Svensson and Yuxuan Xia at Signal processing, Chalmers University of Technology, for making the project possible from the beginning. Also, for teaching us everything we know about multi-object tracking and for providing us with data to try our models out in practice. Hopefully, our thesis and our implementations can help you in your future research.

Camilla Carlsson, Hanna Ekelund, Gothenburg, May 2022

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

ADMM	Alternating Direction Method of Multipliers
ADS	Average direction strategy
GOSPA	Generalized optimal sub-pattern assignment
IP	Integer program
JVC	Jonker-Volgenant-Castanon
LP	Linear Program
MDS	Modified deflected subgradient
MBLP	Mixed-binary linear optimization program
MILP	Mixed-integer linear optimization program
MGT	Modified gradient technique
MOT	Multi-object tracking

Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables that have been used throughout this thesis.

Indices

i,j	Indices for ground truths and estimates, respectively
k	Index for time step
s	Index for iteration

Sets

$\mathcal{I}$	Set of all indices of ground truths and estimates including the indices of the dummy variables, $\{i, j \mid i = 1, ..., n_{\mathcal{X}} + 1,\ j = 1, ..., n_{\mathcal{Y}} + 1\}$
$\hat{\mathcal{I}}$	Set of all indices of ground truths and estimates excluding the indices of the dummy variables, $\{i, j \mid i = 1, ..., n_{\mathcal{X}},\ j = 1, ..., n_{\mathcal{Y}}\}$
$\mathcal{K}$	Set of all time steps, $\{k \mid k = 1, ..., T\}$
$\hat{\mathcal{K}}$	Set of all time steps excluding the last time step T, $\{k \mid k = 1, ..., T - 1\}$
$\mathcal{W}$	Set of all possible assignment matrices
$\overline{\mathcal{W}}$	LP relaxed set of $\mathcal{W}$
$\mathcal{X}$	Set of ground truths
$\mathcal{Y}$	Set of estimates

Parameters

c	Maximum allowed localization error
$d_{k,i,j}$	Element of distance matrix

$n_{\mathcal{X}}$	Number of ground truths
$n_{\mathcal{Y}}$	Number of estimates
p	Penalty coefficient for the error
T	Number of time steps
$\mathbf{x}_{k,i}$	Position of ground truth i at time step k
$\mathbf{y}_{k,j}$	Position of estimate j at time step k
α	Polyak step length parameter
ζ	Relaxation term
μ	Penalty parameter for track switches
ρ	Augmented Lagrangian parameter

Variables

$h_{k,i,j}$	$= 1$ if there is a track switch between time step k and $k+1$ for ground truth i and estimate j; $= 0$ otherwise
$w_{k,i,j}$	$= 1$ if there is an assignment in time step k between ground truth i and estimate j; $= 0$ otherwise
$\gamma_{k,i,j}$	Element k,i,j of a subgradient to the Lagrangian dual function in time step k, ground truth i and estimate j
$\lambda_{k,i,j}$	Lagrangian dual variable corresponding to the relaxed constraint type 1 at time step k, ground truth i and estimate j
$\xi_{k,i,j}$	Lagrangian dual variable corresponding to the relaxed constraint type 2 at time step k, ground truth i and estimate j
$\phi_{k,i,j}$	Slack variable of constraint type 1 at time step k, ground truth i and estimate j
$\psi_{k,i,j}$	Slack variable of constraint type 2 at time step k, ground truth i and estimate j

Contents

1

Introduction

1.1 Multi-object tracking algorithm

A Multi-object tracking algorithm, or *MOT algorithm*, is used to estimate the true position of a set of objects over time. The set of true positions over time of an object is often referred to as a ground truth. The goal of the MOT algorithm is to generate estimates as close to the ground truth as possible.

There are four types of errors that can occur in MOT algorithms. First of all, it can be a distance error. This means that the estimate and the ground truth are not at the same location. It can also happen that the MOT algorithm misses an object, which means that it thinks that a ground truth does not exist at a specific time step. This is referred to as a missed object. If the MOT algorithm instead generates an estimate of a non-existing ground truth, it is referred to as a false object. The final error is called *track switch* and it occurs when an estimate switches which ground truth object it follows and vice versa.

MOT algorithms can be used in multiple areas, for example, in self-driving vehicles for which the MOT algorithms are used for tracking its surroundings. It is important for the MOT algorithms to correctly detect objects. A missed or false object can in the worst case lead to a fatal outcome. Since there are multiple MOT algorithms, it is important to quantify their performances, to be able to compare different algorithms to each other.

MOT algorithms have previously been evaluated efficiently in single time steps using the *GOSPA metric*. A disadvantage with this metric is that it only takes distance, false and missed detected error, but provides no information about track switches.

1.2 Problem formulation

This thesis will explore how an extended version of the GOSPA metric can be solved efficiently. The extended GOSPA metric, developed by A S. Rahmathullah, Á F. García-Fernández, and L. Svensson [1], takes both distance, missed and false errors into consideration, as well as track switches. The model and its variables, sets, and parameters will be presented in this section.

To use this metric, we first denote $n_{\mathcal{X}}$ as the number of unique ground truths, $n_{\mathcal{Y}}$ as

the number of unique estimates, and T as the number of time steps evaluated. We also introduce two dummy variables, one dummy estimate, and one dummy ground truth. Why we need these will be explained later in this section.

Using these three parameters, we define four sets of indices, first

$$\mathcal{I} := \left\{ (i,j) \;\middle|\; i = 1, ..., n_{\mathcal{X}} + 1,\ j = 1, ..., n_{\mathcal{Y}} + 1 \right\} \tag{1.1a}$$

which includes all indices of ground truths and estimates, including the indices of the dummy variables. The second set

$$\hat{\mathcal{I}} := \left\{ (i,j) \;\middle|\; i = 1, ..., n_{\mathcal{X}},\ j = 1, ..., n_{\mathcal{Y}} \right\} \tag{1.1b}$$

is a subset of $\mathcal{I}$, in which the indices of the dummy variables are excluded. The third set

$$\mathcal{K} := \left\{ k \;\middle|\; k = 1, ..., T \right\} \tag{1.2a}$$

includes all time steps. The last index set

$$\hat{\mathcal{K}} := \left\{ k \;\middle|\; k = 1, ..., T - 1 \right\} \tag{1.2b}$$

is a subset of $\mathcal{K}$ where all time step but the last time step T is included.

Let $\mathcal{X}$ define the set of positions $\mathbf{x}_{k,i}$ of all ground truths $i \in \{1, ..., n_{\mathcal{X}}\}$ in every time step $k \in \mathcal{K}$. If the ground truth i does not exist at some time step k, then the position $\mathbf{x}_{k,i}$ equals the empty set $\emptyset$. The set of estimates $\mathcal{Y}$ is defined analogously and define the position of all estimates $j \in \{1, ..., n_{\mathcal{Y}}\}$ in every time step $k \in \mathcal{K}$. Likewise, if the estimate j does not exist at some time step k, then $\mathbf{y}_{k,j} = \emptyset$. Which ground truth i that belongs to estimate j is unknown, and vice versa.

At every time step k, a ground truth i can be assigned to an estimate j. In a perfect case, the number of existing ground truths equals the number of existing estimates in every time step k. Then, each ground truth i can be assigned to an estimate j in every time step k.

Two errors that can occur in a MOT algorithm are false objects and missed objects. If there exists more estimates as compared to the existing ground truths in a time step k, then there exists at least one false object. The meaning of this is that the MOT algorithm estimates there exists an object at a position $\mathbf{y}_{k,j}$ but that actually does not exist. If there instead exists more ground truths as compared to estimates in a time step k, then there exists at least one missed object. The meaning of this is that the MOT algorithm missed to detect a ground truth i, and thereby thinks that an object at a position $\mathbf{x}_{k,i}$ does not exist, while it actually does.

Even if the number of ground truths and estimates are the same at a time step k, there might still exist both missed and false objects. To be able to penalize these types of errors, we introduce one dummy estimate $n_{\mathcal{Y}} + 1$ and one dummy ground truth $n_{\mathcal{X}} + 1$, which may be assigned to multiple ground truths or estimates, respectively. For every ground truth assigned to the dummy estimate $n_{\mathcal{Y}} + 1$, it is

referred to as a missed object. For every estimate assigned to the dummy ground truth, it is referred to as a false object. A dummy estimate may, however, not be assigned to a dummy ground truth.

This problem can be formulated as a type of an assignment problem. We introduce the binary variables $w_{k,i,j}$, $k \in \mathcal{K}, (i,j) \in \mathcal{I}$. Let $w_{k,i,j} = 1$ if there is an assignment in time step k between ground truth i and estimate j, otherwise $w_{k,i,j} = 0$.

In every time step k, every ground truth is assigned to one estimate, or dummy estimate. Also, every estimate is assigned to one ground truth, or dummy ground truth. This can be described in mathematical terms as, for each $k \in \mathcal{K}$,

$$\sum_{i=1}^{n_{\mathcal{X}}+1} w_{k,i,j} = 1, \qquad j \in \{1, ..., n_{\mathcal{Y}}\}, \tag{1.3a}$$

$$\sum_{j=1}^{n_{\mathcal{Y}}+1} w_{k,i,j} = 1, \qquad i \in \{1, ..., n_{\mathcal{X}}\}, \tag{1.3b}$$

$$w_{k,n_{\mathcal{X}}+1,n_{\mathcal{Y}}+1} = 0, \tag{1.3c}$$

$$w_{k,i,j} \in \{0,1\}, \qquad (i,j) \in \mathcal{I}. \tag{1.3d}$$

Note that these assignment constraints are separated over each time step k. Therefore, we define the matrix of binary assignment variables in a given time step k as W_k, $W_k = (w_{k,i,j} \mid (i,j) \in \mathcal{I})$, $k \in \mathcal{K}$. The set of feasible assignments in any given time step is then expressed as

$$\mathcal{W} := \left\{ W_k \in \mathbb{R}^{(n_{\mathcal{X}}+1)\times(n_{\mathcal{Y}}+1)} \;\middle|\; \text{the constraints (1.3) hold} \right\}. \tag{1.4}$$

For simplicity, we let W denote the compound of all assignment matrices W_k, i.e., $W = (W_k \mid k \in \mathcal{K})$.

The distance matrix D, is a $T \times (n_{\mathcal{X}} + 1) \times (n_{\mathcal{Y}} + 1)$-matrix. If ground truth i and estimate j exist in time step k; the element $d_{k,i,j}$ is defined as the minimum of the cut-off parameter c to the power of p, where p is a positive integer, and the L_p-norm of the distance between the ground truth i and the estimate j, to the power of p. If neither ground truth i nor estimate j exist at time step k, the element $d_{k,i,j}$ is equal to zero. If exactly one of ground truth i or estimate j exists at time step k, the element $d_{k,i,j}$ equals $c^p/2$. In mathematical notation, the distance matrix is hence defined as

$$d_{k,i,j} = \begin{cases} \min\left\{c^p, \|\mathbf{x}_{k,i} - \mathbf{y}_{k,j}\|_p^p\right\}, & \text{if } \mathbf{x}_{k,i} \neq \emptyset \text{ and } \mathbf{y}_{k,j} \neq \emptyset, \\ 0, & \text{if } \mathbf{x}_{k,i} = \mathbf{y}_{k,j} = \emptyset, \\ \frac{c^p}{2}, & \text{otherwise} \end{cases} \qquad k \in \mathcal{K},\ (i,j) \in \mathcal{I}. \tag{1.5}$$

Observe that $\mathbf{x}_{k,n_{\mathcal{X}}+1} = \mathbf{y}_{k,n_{\mathcal{Y}}+1} = \emptyset$, since neither of the dummy ground truth nor dummy estimate exist at any time step $k \in \mathcal{K}$.

If both ground truth i and estimate j exists, and the distance is smaller than c^p, $d_{k,i,j}$ represents the localization error. If the distance is larger or equal than c^p, and

ground truth i and estimate j are assigned to each other in time step k, the ground truth i is interpreted as a missed object and the estimate j is interpreted as a false object in time step k.

If exactly one of ground truth i or estimate j exists at time step k, and these two are assigned to each other, then it is interpreted as a missed or false cost, depending on whether ground truth i or estimate j exists.

Each ground truth i does not have to be assigned to the same estimate j in each time step k. If an assignment between ground truth i and estimate j between time step k and $k+1$ are different, hence $w_{k,i,j} \neq w_{k+1,i,j}$, it is referred to as a track switch. The interpretation of this being that the MOT algorithm thinks a ground truth i is different objects in time step k and $k+1$.

For example, if two cars, a red and a blue car, are driving parallel to each other in two lanes. In the first two time steps, we estimate the red car to be red, and the blue car to be blue. In between time step two and time step three, the blue and the red car swap lanes. In the last two time steps, the MOT algorithm thinks the two cars are in their original lanes and do not think the lane swap occurred. Therefore, in the two last time steps, we estimate the red car to be blue and the blue car to be red, see Fig. 1.1.

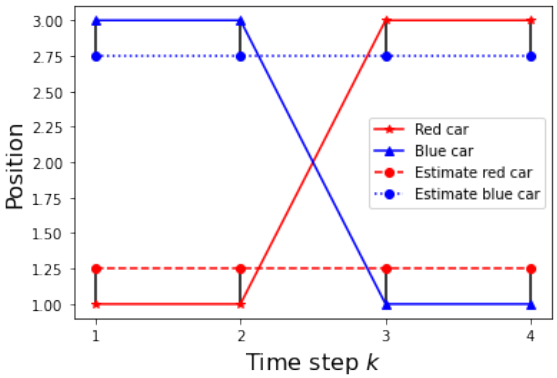


Figure 1.1: Illustration of a track switch. The red car is estimated to be red in time step one and two, and blue in time step three and four. The blue car is estimated to be blue in time step one and time step two, and red in time step three and time step four.

If a ground truth i changes from being assigned to the dummy estimate $n_y + 1$ to an estimate j in the next time step or vice versa, this is referred to as a *half switch*. The meaning of this is that a ground truth i goes from being unassigned to being assigned in the next time step. Analogously, a half switch occurs if an estimate j changes from being unassigned to being assigned, or the other way around, in the next time step.

To evaluate the performance of a MOT algorithm, we need to solve T sequentially connected assignment problems, such that $W_k \in \mathcal{W}, k \in \mathcal{K}$ and penalize any time an assignment changes between time step k and $k+1$. The number of half track

switches is calculated as the absolute difference of $w_{k,i,j}$ and $w_{k+1,i,j}$, $(i,j) \in \hat{\mathcal{I}}$ and $k \in \hat{\mathcal{K}}$. Two half switches equal one track switch. Every track switch is penalized with μ^p, the higher the value, the more track switches will be avoided.

For some set of ground truths $\mathcal{X}$ and set of estimates $\mathcal{Y}$, we can formulate the optimization problem as follow

$$\underset{W}{\text{minimize}} \quad \left(\sum_{k=1}^{T} \sum_{i=1}^{n_{\mathcal{X}}+1} \sum_{j=1}^{n_{\mathcal{Y}}+1} d_{k,i,j} w_{k,i,j} + \frac{\mu^p}{2} \sum_{k=1}^{T-1} \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \left| w_{k,i,j} - w_{k+1,i,j} \right| \right)^{\frac{1}{p}} \tag{1.6a}$$

$$\text{subject to} \quad W_k \in \mathcal{W}, \quad k \in \mathcal{K}. \tag{1.6b}$$

Since the objective function (1.6a) is not linear due to the absolute value of the difference of variables, we want to rewrite it into a linear function to be solved easier. To obtain a linear objective function, we introduce help variables $h_{k,i,j}$ to represent the number of half track switches to replace the absolute value in the objective function. We also introduce two constraints for each help variable $h_{k,i,j}$. First, $h_{k,i,j}$ needs to be larger or equal than the difference between the solution of the current time step assignment solution for $w_{k,i,j}$ and the solution of the forward time step assignment solution, $w_{k+1,i,j}$. Second, $h_{k,i,j}$ needs to be larger or equal than the difference between the solution of the forward time step assignment solution for $w_{k+1,i,j}$ and the solution of the current time step assignment solution, $w_{k,i,j}$. This optimization problem will be formulated as,

$$z^\star := \underset{W,H}{\text{minimum}} \quad \left(\sum_{k=1}^{T} \sum_{i=1}^{n_{\mathcal{X}}+1} \sum_{j=1}^{n_{\mathcal{Y}}+1} d_{k,i,j} w_{k,i,j} + \frac{\mu^p}{2} \sum_{k=1}^{T-1} \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} h_{k,i,j} \right)^{\frac{1}{p}} \tag{1.7a}$$

$$\text{subject to} \quad W \in \mathcal{W}, \tag{1.7b}$$

$$h_{k,i,j} \geq w_{k,i,j} - w_{k+1,i,j}, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, \tag{1.7c}$$

$$h_{k,i,j} \geq w_{k+1,i,j} - w_{k,i,j}, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}. \tag{1.7d}$$

Since the objective function always is greater or equal than zero and p is a positive integer, this problem can be treated as a *mixed-binary linear optimization problem* (MBLP); since all the constraints as well as the objective function are defined by linear forms and the variables $w_{k,i,j}$ are binary while $h_{k,i,j}$ are continuous variables.

1.3 Aim and limitations

The aim with this project is to develop an efficient optimization algorithm for solving the optimization problem (1.7), to quantify the performance of a MOT algorithm over a finite sequence of time steps. The performance will take into consideration, the distance between sets of points in the ground truth and the estimate, as well as penalties for missed objects, false objects, and track switches. Lower optimal objective values will imply better MOT algorithms. The goal is to be able to use the algorithm to quickly get a measure of the performance of a MOT algorithm in order to compare it with other MOT algorithms.

The project will only focus on the development of the optimization algorithm and will not be a MOT algorithm itself, it will only quantify the performance of MOT algorithms. The project will not explore how MOT algorithms work, only use the two sets $\mathcal{X}$ and $\mathcal{Y}$ together with some user-specific parameters $c, p,$ and μ, to evaluate the performance of the MOT algorithm generating the set $\mathcal{Y}$.

The project has implemented two optimization algorithms, a multi-block *alternating direction method of multipliers* (ADMM) and *modified deflected subgradient* method (MDS) applied to a Lagrangian dual of the model (1.7).

2

Theory

2.1 General integer optimization

A general *integer program* (IP) can be written as

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & \boldsymbol{c}^\top \boldsymbol{x}, && \text{(2.1a)} \\ \text{subject to} \quad & A\boldsymbol{x} \geq \boldsymbol{b}, && \text{(2.1b)} \\ & \boldsymbol{x} \in \mathbb{Z}^n, && \text{(2.1c)} \end{aligned}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$ and $A \in \mathbb{R}^{n \times m}$. General integer programs are $\mathcal{NP}$-hard problems and are classified as hard problems since they cannot be solved within polynomial time. If the matrix A has only integer elements, then it is called an integral matrix.

One can, however, use LP relaxation in order to transform the IP into an LP. This is advantageous since linear problems are easy and can be solved in polynomial time. The only difference between an IP and LP is that the constraint (2.1c) is changed to $\boldsymbol{x} \in \mathbb{R}^n$, while the objective function and constraint (2.1b) remain the same. A *mixed-integer linear optimization program* (MILP) is a combination of an IP and an LP, where some of the variables are integers and the rest are continuous [2]. A special case of these problems is when all integer variables are binary, then it is called an MBLP.

An important property an LP possesses is that the optimal point is found in an extreme point of the feasible set, as long as the feasible set is bounded and non-empty. The solution to the LP-relaxed problem might not be optimal in the original MILP, but is only guaranteed to yield a lower bound, unless the feasible set has the *integrality property*.

Theorem 2.1.1 (Hoffman and Kruskal, see [3, Theorem 4.4]) *"Let A be an $m \times n$ integral matrix. The polyhedron $\boldsymbol{x} : A\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \geq 0$ is integral for every $\boldsymbol{b} \in \mathbb{Z}_m$ if and only if A is totally unimodular."*

The feasible set is the polyhedron spanned by the constraints $A\boldsymbol{x} \geq \boldsymbol{b}$, $\boldsymbol{x} \geq 0^n$ and it has the integrality property if and only if each extreme point of the polyhedron is integer valued. A set can be shown to have the integrality property by utilizing Theorem 2.1.1. The theorem states that the matrix A has to be totally unimodular and by definition, a totally unimodular matrix is defined as

"A matrix A is totally unimodular if every square submatrix has determinant $0, \pm 1$." ([3, p. 131])

2.2 Lagrangian relaxation

Assume one have a MILP problem where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$ and $A \in \mathbb{R}^{n \times m}$ on the following form

$$z^\star := \underset{\mathbf{x}}{\text{minimum}} \quad \mathbf{c}^\top \mathbf{x}, \tag{2.2a}$$

$$\text{subject to} \quad A\mathbf{x} \geq \mathbf{b}, \tag{2.2b}$$

$$\mathbf{x} \in \mathcal{S}, \tag{2.2c}$$

where the constraints (2.2b) are assumed to complicate the problem. The easy constraints form the set $\mathcal{S}$ and are represented in (2.2c). When applying Lagrangian relaxation to the complicating constraints (2.2b), the constraints are moved to the objective function and each of them multiplied by an individual Lagrangian multiplier $u_i \in \mathbb{R}$. The *Lagrangian function* is defined as [2, Chapter 17],

$$\mathcal{L}(\mathbf{x}, \mathbf{u}) := \mathbf{c}^\top \mathbf{x} + \sum_{i=1}^{m} u_i \Big(b_i - \boldsymbol{A}_i \mathbf{x} \Big), \quad \mathbf{x} \in \mathcal{S}. \tag{2.3}$$

The *Lagrangian dual function* is then defined as

$$g(\mathbf{u}) := \underset{\mathbf{x} \in \mathcal{S}}{\text{minimum}} \left\{ \mathbf{c}^\top \mathbf{x} + \sum_{i=1}^{m} u_i \Big(b_i - \boldsymbol{A}_i \mathbf{x} \Big) \right\}. \tag{2.4}$$

The Lagrangian dual function will generate a lower bound on the primal optimal value $z^\star$ for any $\boldsymbol{u} \geq \mathbf{0}$ [2, Chapter 17].

If the feasible set $\mathcal{S}$ is non-convex due to integer variables $\mathbf{x}$, then one can use LP-relaxation to transform the problem into a convex set. This can transform the problem into a convex optimization problem which can be solved in polynomial time [4]. The LP relaxed set we call $\bar{\mathcal{S}}$. The Lagrangian dual function is a concave function [2, Chapter 17], constructed out of linear segments, one for each extreme point in the feasible set $\mathcal{S}$. The *Lagrangian dual problem* is to maximize the Lagrangian dual function over non-negative values of the vector $\mathbf{u}$, to yield an as good lower bound as possible, i.e.,

$$g^\star := \underset{\mathbf{u} \geq \mathbf{0}}{\text{maximum}} \left\{ g(\mathbf{u}) \right\}. \tag{2.5}$$

The advantage of relaxing an optimization problem is that the Lagrangian dual function is often separable and its evaluation can be divided into several smaller subproblems, which can be solved individually instead of as one large problem. If the convex hull of the set $\mathcal{S}$, $\text{conv}(\mathcal{S})$ equals the set $\bar{\mathcal{S}}$, then the lower bound from the Lagrangian dual problem $g^\star$ will be equal to the primal optimal value $z^\star$, namely, we

will have *strong duality* $g^\star = z^\star$ [2]. This is true for all pure linear programs where the feasible set $\mathcal{S}$ is convex. In other cases where $\bar{\mathcal{S}} \supset \text{conv}(\mathcal{S})$, strong duality does not have to hold. The Lagrangian dual problem $g^\star$ can therefore provide a lower bound which is smaller than the primal optimal value $z^\star$, as $g^\star \leq z^\star$. In this case, most commonly we will have a *duality gap* between $z^\star$ and $g^\star$. This indicates that the optimal solution to the Lagrangian dual problem is not an integer point. If we do not have a duality gap when $\bar{\mathcal{S}} \supset \text{conv}(\mathcal{S})$, then the optimal solution is integer in both the primal problem and the dual problem.

2.3 Integrality property

Regarding the integrality property of the evaluation problem (1.7), we could neither prove nor disprove that it has the integrality property. However from what we know about the problem a conjecture was made.

Conjecture 2.3.1 *The MILP problem* (1.7) *has the integrality property and the solution to the LP relaxation of* (1.7) *has integer extreme values and is a solution to the non-relaxed problem as well.*

The reason we strongly believe that we have integrality property is that all the solutions found during this thesis were integer. Despite the fact that we let a computer randomly generate and solve instances of (1.7) during a main part of our Masters project, we could not construct a problem instance with non-integer solutions to the LP relaxation.

Another reason which supports the conjecture that the evaluation problem (1.7) has integer optimal solutions is the fact that it is made up of multiple assignment problems which are known to have the integrality property. The evaluation problem is made up of connected assignment problems and the connected part might destroy the integrality property. We will try to use Theorem 2.1.1 in order to prove that our problem has the integrality property. Since we know that the constraints from the assignment problems define a totally unimodular matrix A, we focus on the additional constraints corresponding to the connected part. The connected constraints are formulated as

$$w_{k,i,j} - w_{k+1,i,j} \leq h_{k,i,j}, \quad k \in \hat{\mathcal{K}}, \quad i,j \in \hat{\mathcal{I}} \tag{2.6a}$$

$$w_{k+1,i,j} - w_{k,i,j} \leq h_{k,i,j}, \quad k \in \hat{\mathcal{K}}, \quad i,j \in \hat{\mathcal{I}} \tag{2.6b}$$

and from this, we can construct the submatrix corresponding to these two constraints for some fixed k, i, and j. We then get

$$\begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \end{bmatrix} \tag{2.7}$$

where the first column corresponds to the variable $w_{k,i,j}$, the second to $w_{k+1,i,j}$ and the third to $h_{k,i,j}$. There are three square subdeterminants to this matrix, namely

$$\begin{vmatrix} 1 & -1 \\ -1 & 1 \end{vmatrix} = 0, \quad \begin{vmatrix} 1 & -1 \\ -1 & -1 \end{vmatrix} = -2, \quad \begin{vmatrix} -1 & -1 \\ 1 & -1 \end{vmatrix} = 2, \tag{2.8}$$

which means that the matrix is not totally unimodular since some subdeterminant does not equal ± 1 or 0.

Theorem 2.1.1 states that the polyhedron is integer for every integer vector $\boldsymbol{b}$. Although, for our case, we only care about a specific $\boldsymbol{b}$. We know that the elements in $\boldsymbol{b}$ values corresponding to the constraints (2.6) will all be zero. By this, we believe it is possible to construct a problem that has the integrality property without the matrix A being totally unimodular.

The proof of Theorem 2.1.1 utilizes the fact that at an extreme point to the polyhedron $\{x \geq 0^n : A\boldsymbol{x} \leq \boldsymbol{b}\}$ satisfies n linearly independent inequalities that span the polyhedron at equality [3, p. 132]. The n equations can then be written as $C\boldsymbol{x} = \boldsymbol{d}$ and solved as a linear system $\bar{\boldsymbol{x}} = C^{-1}\boldsymbol{d}$, where $\bar{\boldsymbol{x}}$ is the coordinates of an extreme point of the polyhedron and the n inequalities $C\boldsymbol{x} \leq \boldsymbol{d}$ are satisfied at equality. The proof of Theorem 2.1.1 then uses the fact that A, and thereby also C, is totally unimodular to show that C^{-1} is integral, meaning that each of its elements is an integer. This does not hold for our problem but we know that some elements in $\boldsymbol{d}$ equal 0 and that any real number multiplied by 0 is 0 and is therefore integer. We wish to show that all non-integer elements of C^{-1} will be multiplied by some 0 element in the $\boldsymbol{d}$ vector when solving for $\bar{\boldsymbol{x}}$. Our problem will look like

$$A = \begin{bmatrix} F & 0 \\ I & 0 \\ 0 & I \\ & E \end{bmatrix} \tag{2.9}$$

where F corresponds to the assignment constraints which only depend on the W variables and E to the connected constraints (2.6). We can only guarantee that part of the matrix A is totally unimodular, we also know that the elements in the $\boldsymbol{b}$ vector corresponding to the matrix E will be 0, and the matrix E is the only part that is not totally unimodular. We define $C\boldsymbol{x} = \boldsymbol{d}$ to be n linearly independent inequalities from our polyhedron $A\boldsymbol{x} \leq \boldsymbol{b}$ satisfied at equality. Our conjecture is that all non-integer elements in C^{-1} will be multiplied by one of the elements in $\boldsymbol{d}$ that we know is 0.

To explain why we believe this holds and why this gives us integer solutions we have to take a closer look at how C^{-1} interact with $\boldsymbol{d}$ and how C and C^{-1} correspond to each other. If we assume that all integer elements in C^{-1} are multiplied by some integer value in $\boldsymbol{d}$ and all non-integer elements in C^{-1} by 0, then it follows that $\bar{\boldsymbol{x}}$ is integer since the sum of some integers and zero is integer. As stated previously $\bar{\boldsymbol{x}}$ will be a extreme point to the polyhedron $A\boldsymbol{x} \leq \boldsymbol{b}$ and if this holds for every linearly independent combination of n constraints then it holds that every extreme point to $A\boldsymbol{x} \leq \boldsymbol{b}$ is integer and therefore the polyhedron $A\boldsymbol{x} \leq \boldsymbol{b}$ is integral [3, Theorem 4.1].

Now if we take a closer look at how C and C^{-1} interact with each other, by Cramer's rule C^{-1} is equal to the adjugate matrix of C divided by the determinant of C [3]. The adjugate matrix is defined as the transpose of the cofactor matrix which in turn is defined by the minor of C. Let M_{ij} denote the (i, j)-minor of C, which is defined as the determinant of the $(n-1) \times (n-1)$ matrix of C where row i and column

j are removed. The cofactor matrix of C is $[(-1)^{i+j}M_{ij}]_{1\leq i,j\leq n}$ and the adjugate of C then becomes $[(-1)^{i+j}M_{ji}]_{1\leq i,j\leq n}$. Since the matrix C is a square submatrix of A and we wish to keep track of the rows from submatrix E, we introduce different notations depending on which submatrix a row comes from. We denote the rows of matrix C that comes the submatrix E as $C(E)$ and the rows from matrix A that are not in E as $C(A \setminus E)$. For simplicity, all rows from the matrix E are placed at the bottom and we can write

$$C = \begin{bmatrix} C(A \setminus E) \\ C(E) \end{bmatrix}, \quad \boldsymbol{d} = \begin{bmatrix} d_1 \\ \vdots \\ d_m \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{2.10}$$

where the matrix C consists of $n - m$ rows from the matrix E and m from the matrix $A \setminus E$. Then we denote the (i, j)-minor as $M(E)_{ij}$ if row i comes from the matrix E and $M(A \setminus E)_{ij}$ otherwise. Then we can write C^{-1} in terms of $M(E)$ and $M(A \setminus E)$ as

$$C^{-1} = \frac{1}{\det(C)} \begin{bmatrix} M(A \setminus E)_{11} & \dots & (-1)^{1+n}M(A \setminus E)_{1n} \\ \vdots & & \vdots \\ (-1)^{m+1}M(A \setminus E)_{m1} & \dots & (-1)^{m+n}M(A \setminus E)_{mn} \\ (-1)^{(m+1)+1}M(E)_{(m+1)1} & \dots & (-1)^{(m+1)+n}M(E)_{(m+1)n} \\ \vdots & & \vdots \\ (-1)^{n+1}M(E)_{n1} & \dots & M(E)_{nn} \end{bmatrix}^\intercal \tag{2.11}$$

where $\det(C)$ is the determinant of C. Finally we can solve for $\bar{\boldsymbol{x}}$ and get

$$\begin{aligned} \bar{\boldsymbol{x}} =& \frac{1}{\det(C)} \begin{bmatrix} \sum_{i=1}^{m}(-1)^{i+1}d_i M(A \setminus E)_{i1} + \sum_{j=m+1}^{n}(-1)^{j+1} \cdot 0 \cdot M(E)_{j1} \\ \vdots \\ \sum_{i=1}^{m}(-1)^{i+n}d_i M(A \setminus E)_{in} + \sum_{j=m+1}^{n}(-1)^{j+n} \cdot 0 \cdot M(E)_{jn} \end{bmatrix} \\ =& \frac{1}{\det(C)} \begin{bmatrix} \sum_{i=1}^{m}(-1)^{i+1}d_i M(A \setminus E)_{i1} + 0 \\ \vdots \\ \sum_{i=1}^{m}(-1)^{i+n}d_i M(A \setminus E)_{in} + 0 \end{bmatrix} \end{aligned} \tag{2.12}$$

where we can see that as long as $\sum_{i=1}^{m}(-1)^{i+j}d_i M(A \setminus E)_{ij}$ is divisible by $\det(C)$ for all $j = 1, .., n$ then $\bar{\boldsymbol{x}}$ will be integer. This is easy to prove if every $M(A \setminus E)_{ij}$ is divisible by $\det(C)$, since we know that each d_i is integer. This is easy for one specific instance but it has to be done for every combination of n linearly independent rows from the matrix A in order to prove that the polyhedron $A\boldsymbol{x} \leq \boldsymbol{b}$ is integral. Then one needs to prove that it holds for any number of ground truths, estimates, and time steps for the problem (1.7) to have integer properties for every instance. One might be able to prove this by induction, but even proving that $M(A \setminus E)_{ij}$ is divisible by the determinant of C is hard. Our understanding is that all the subdeterminants of C that are not equal to 0 or ± 1 have at least 1 row that is from the matrix E.

Therefore $M(A \setminus E)_{ij}$ is probably divisible by the determinant of C, since removing a row that is from the matrix $A \setminus E$ should not change the determinant, however, removing a column might change the value.

2.4 Linearization of convex quadratic functions

For some special cases of a quadratic integer program, it is possible to replace the quadratic term with a linear term (see [5]). Assume we have a feasible set $\mathcal{S}$, including all binary vectors $\boldsymbol{x} \in \mathbb{B}^n$, fulfilling some constraints $B\boldsymbol{x} = \boldsymbol{b}$

$$\mathcal{S} := \{\boldsymbol{x} \in \{0,1\}^n \mid B\boldsymbol{x} = \boldsymbol{b}\}. \tag{2.13}$$

The quadratic optimization problem can be expressed in vector form, where Q is an $\mathbb{R}^{n\times n}$-matrix, where the goal is to minimize the quadratic function over some set $\mathcal{S}$, as

$$\underset{\boldsymbol{x}\in\mathcal{S}}{\text{minimize}}\ \boldsymbol{q}^\top\boldsymbol{x} + \boldsymbol{x}^\top Q\boldsymbol{x}. \tag{2.14}$$

The corresponding linear problem can be expressed in vector form, where $\boldsymbol{c} \in \mathbb{R}^n$, where the goal is to minimize the linear function over some set $\mathcal{S}$. The binary linear optimization problem can be formulated as

$$\underset{\boldsymbol{x}\in\mathcal{S}}{\text{minimize}}\ \boldsymbol{c}^\top\boldsymbol{x}. \tag{2.15}$$

The quadratic optimization problem is equivalent to the linear optimization problem, if and only if,

$$\boldsymbol{q}^\top\boldsymbol{x} + \boldsymbol{x}^\top Q\boldsymbol{x} = \boldsymbol{c}^\top\boldsymbol{x}, \quad \forall \boldsymbol{x} \in \mathcal{S}, \tag{2.16}$$

see [5]. This is true if Q is a diagonal matrix where $\text{Diag}(\boldsymbol{q}) + Q = \text{Diag}(\boldsymbol{c})$.

2.5 Alternating direction method of multipliers (ADMM)

The alternating direction method of multipliers (ADMM) (see [6]) is an algorithm that solves optimization problems where the objective function is separable with respect to the variables, but the constraints are not. It utilizes the augmented Lagrangian (see [7]) in order to relax the constraints and solves the problem iteratively, and sequentially with respect to one variable vector at a time. Specifically it considers optimization problems on the form

$$z^\star := \underset{x_1,\ldots,x_n}{\text{minimum}} \quad \sum_{i=1}^{n} f_i(x_i), \tag{2.17a}$$

$$\text{subject to} \quad \sum_{i=1}^{n} A_i x_i = \boldsymbol{c} \tag{2.17b}$$

where $x_i \in \mathbb{R}^{m_i}$ are variables, and $A_i \in \mathbb{R}^{p \times m_i}$ and $\boldsymbol{c} \in \mathbb{R}^p$ are constant matrices and vectors, respectively. The problem consists of minimizing the sum of n functions f_i that are all convex and depend only on the corresponding variable vector x_i, subject to p constraints that depend on multiple variables x_i. This problem is solved with a multi-block ADMM where $n > 2$, while if $n = 2$ it can be solved with a two-block ADMM. The distinction is made since the convergence analysis differs between the two cases. The problem may also include constraints that only depend on one of the variables x_i, like $x_i \in \mathcal{X}_i$. Such constraints are included in the function f_i as an indicator function of the set $\mathcal{X}_i$. The set $\mathcal{X}_i$ must be convex in order for its indicator function and thereby the function f_i to be convex. The algorithm works for any finite number of variables as long as they are separable in the objective. The number of relaxed constraints is not restricted either.

The augmented Lagrangian function is formulated as

$$L_\rho(x_1, ..., x_n, \lambda) = \sum_{i=1}^{n} f_i(x_i) + \lambda^\top \left(\sum_{i=1}^{n} A_i x_i - \boldsymbol{c} \right) + \frac{\rho}{2} \left\| \sum_{i=1}^{n} A_i x_i - \boldsymbol{c} \right\|_2^2, \tag{2.18}$$

where $\lambda \in \mathbb{R}^p$ is the vector of dual variables and $\rho > 0$ is the augmented Lagrangian parameter. The ADMM iterations start with some initialization $\boldsymbol{x}^0$ and $s = 0$ and are formulated from the augmented Lagrangian and consist of the steps

$$x_1^{s+1} := \underset{x_1}{\operatorname{argmin}}\, L_\rho(x_1, x_2^s, ..., x_n^s, \lambda^s), \tag{2.19a}$$

$$x_i^{s+1} := \underset{x_i}{\operatorname{argmin}}\, L_\rho(x_1^{s+1}, ..., x_{i-1}^{s+1}, x_i, x_{i+1}^s, ..., x_n^s, \lambda^s), \quad i = 2, ..., n-1, \tag{2.19b}$$

$$x_n^{s+1} := \underset{x_n}{\operatorname{argmin}}\, L_\rho(x_1^{s+1}, ..., x_{n-1}^{s+1}, x_n, \lambda^s), \tag{2.19c}$$

$$\lambda^{s+1} := \lambda^s + \rho \left(\sum_{i=1}^{n} A_i x_i^{s+1} - \boldsymbol{c} \right), \tag{2.19d}$$

where s denotes the iteration index. The algorithm consists of n primal minimization steps and one dual variable update.

In order to guarantee convergence of two-block ADMM (i.e., for $n = 2$) two assumptions must be fulfilled, one about the functions f_i and the other about the augmented Lagrangian. To guarantee convergence of multi-block ADMM, the same two conditions must be fulfilled as well as some additional conditions.

Assumption 2.5.1 (see [6, Assumption 1]) *"The (extended-real-valued) functions $f_i : \mathbb{R}^{m_i} \to \mathbb{R} \cup \{\infty\}, \quad i = 1, ..., n$ are closed, proper and convex."*

Assumption 2.5.1 implies that all subproblems corresponding to updating the x_i variables are solvable, meaning, there exist some values of x_i, i=1, ..., n, that minimize the augmented Lagrangian L_ρ. The solution does not need to be unique, meaning there can be multiple values of x_i which all minimize the augmented Lagrangian. This assumption allows the functions f_i to be indicator functions over sets $\mathcal{X}_i$, as long as the sets $\mathcal{X}_i$ are non-empty and convex. The subproblem corresponding to updating the value of the variable x_i will then be a minimization problem over the set $\mathcal{X}_i$.

Assumption 2.5.2 (see [6, Assumption 2]) *"The unaugmented Lagrangian L_0 has a saddle point."*

Assumption 2.5.2 states that the unaugmented Lagrangian

$$L_0(x_1, ..., x_n, \lambda) = \sum_{i=1}^{n} f_i(x_i) + \lambda \left(\sum_{i=1}^{n} A_i x_i - \boldsymbol{c} \right) \tag{2.20}$$

has a saddle point, which means that there exists some point $(x_1^\star, ..., x_n^\star, \lambda^\star)$ such that

$$L_0(x_1^\star, ..., x_n^\star, \lambda) \leq L_0(x_1^\star, ..., x_n^\star, \lambda^\star) \leq L_0(x_1, ..., x_n, \lambda^\star) \tag{2.21}$$

holds for all $(x_1, ..., x_n, \lambda) \in \left(\bigtimes_{i=1}^{n} \mathbb{R}^{m_i} \right) \times \mathbb{R}^p$. Assumptions 2.5.1 and 2.5.2 together imply that the value $L_0(x_1^\star, ..., x_n^\star, \lambda^\star)$ is finite for any saddle point $(x_1^\star, ..., x_n^\star, \lambda^\star)$ and that the primal point $(x_1^\star, ..., x_n^\star)$ solves (2.17).

Given that Assumptions 2.5.1 and 2.5.2 hold then the two-block ADMM iterations satisfy residual, objective, and dual variable convergence (see [6]). With residual (r^s), objective and dual variable convergence, this means that

$$r^s \to 0 \quad \text{as } s \to \infty; \tag{2.22a}$$

$$\sum_{i=1}^{n} f_i(x_i^s) \to z^\star \quad \text{as } s \to \infty; \tag{2.22b}$$

$$\lambda^s \to \lambda^\star \quad \text{as } s \to \infty \tag{2.22c}$$

holds. In order to guarantee that multi-block ADMM converges a sufficient condition is defined as

> "There exists two integers i and j such that any two matrices in the sets $\{A_i, A_{i+1}, ..., A_{i+j}\}$ and $\{A_{i+j+1}, A_{i+j+2}, ..., A_m, A_1, A_2, ..., A_{i-1}\}$ are orthogonal". ([8, p. 22])

Here A_i represents the constraint matrices in (2.17) and m the number of variable vectors x_i. This states that if the A_i matrices can be split into two sets such that each matrix in either set is orthogonal with every other matrix in the same set, then multi-block ADMM will converge. This is a sufficient condition, but not necessary, which means that even if this does not hold then the multi-block ADMM might still converge. On the other hand, there exists a version of ADMM where the step size ρ is small and a relaxation term $\zeta > 0$ is introduced in the dual variable update step (2.19d) in order to get

$$\lambda^{s+1} := \lambda^s + \rho\zeta \left(\sum_{i=1}^{n} A_i x_i^{s+1} - \boldsymbol{c} \right). \tag{2.23}$$

For two-block ADMM the convergence can be guaranteed for every problem instance, for some $\zeta \in (0, \frac{\sqrt{5}+1}{2})$, however, for multi-block ADMM there exists no such range for ζ such that every problem is guaranteed to converge (see [8]). However, for a specific problem, there might exist a range for ζ such that the problem converges.

The optimality conditions and stopping criterion for the ADMM problem (2.17) are formulated from the primal and dual feasibility conditions, as,

$$\sum_{i=1}^{n} A_i x_i^\star - \boldsymbol{c} = 0 \tag{2.24}$$

and

$$0 \in \partial f_i(x_i^\star) + A_i \lambda^\star, \quad i = 1, ..., n, \tag{2.25}$$

respectively. The primal feasibility states that the solution has to be feasible in the primal problem. The primal residual is constructed as $r^s := \sum_{i=1}^{n} A_i x_i^s - \boldsymbol{c}$. From the ADMM update (2.19c) it is know that x_n^{s+1} minimizes $L_\rho(x_1^{s+1}, ..., x_{n-1}^{s+1}, x_n, \lambda^s)$ and therefore

$$\begin{aligned} 0 &\in \partial f_n(x_n^{s+1}) + A_n^\intercal \lambda^s + \rho A_n^\intercal \left(\sum_{j=1}^{n} A_j x_j^{s+1} - \boldsymbol{c} \right) \\ &= \partial f_n(x_n^{s+1}) + A_n^\intercal \lambda^s + \rho A_n^\intercal r^{s+1} \\ &= \partial f_n(x_n^{s+1}) + A_n^\intercal \lambda^{s+1}. \end{aligned} \tag{2.26}$$

This will always satisfy the dual feasibility (2.25) for $i = n$.

Similarly, from the update (2.19b) it is known that x_i^{s+1} minimizes $L_\rho(x_1^{s+1}, ..., x_{i-1}^{s+1}, x_i, x_{i+1}^s, ..., x_n^s, \lambda^s)$, $\quad i = 2, ..., n-1$, and therefore

$$\begin{aligned} 0 &\in \partial f_i(x_i^{s+1}) + A_i^\intercal \lambda^s + \rho A_i^\intercal \left(\sum_{j=1}^{i} A_j x_j^{s+1} + \sum_{j=i+1}^{n} A_j x_j^s - \boldsymbol{c} \right) \\ &= \partial f_i(x_i^{s+1}) + A_i^\intercal \lambda^s + \rho A_i^\intercal r^{s+1} + \rho A_i^\intercal \sum_{j=i+1}^{n} A_j (x_j^s - x_j^{s+1}) \\ &= \partial f_i(x_i^{s+1}) + A_i^\intercal \lambda^{s+1} + \rho A_i^\intercal \sum_{j=i+1}^{n} A_j (x_j^s - x_j^{s+1}). \end{aligned} \tag{2.27}$$

This is not guaranteed to satisfy the dual feasibility (2.25), however it will be satisfied if the quantity $q_i^{s+1} := \rho \sum_{j=i+1}^{n} A_j (x_j^s - x_j^{s+1})$, $i = 2, ..., n-1$ is 0. The quantity q_i^{s+1} is the dual residual for dual feasibility condition $0 \in \partial f_i(x_i^\star) + A_i \lambda^\star$ at time iteration $s + 1$.

Finally, from the update (2.19a) we have that x_1^{s+1} minimizes $L_\rho(x_1, x_2^s, ..., x_n^s, \lambda^s)$ and

$$\begin{aligned} 0 &\in \partial f_1(x_1^{s+1}) + A_1^\intercal \lambda^s + \rho A_1^\intercal \left(A_1 x_1^{s+1} + \sum_{j=2}^{n} A_j x_j^s - \boldsymbol{c} \right) \\ &= \partial f_1(x_1^{s+1}) + A_1^\intercal \lambda^s + \rho A_1^\intercal r^{s+1} + \rho A_1^\intercal \sum_{j=2}^{n} A_j (x_j^s - x_j^{s+1}) \\ &= \partial f_1(x_1^{s+1}) + A_1^\intercal \lambda^{s+1} + \rho A_1^\intercal \sum_{j=2}^{n} A_j (x_j^s - x_j^{s+1}). \end{aligned} \tag{2.28}$$

As before, this is not guaranteed to satisfy the dual feasibility in (2.25) for $i = 1$, however if the quantity $q_1^{s+1} := \rho \sum_{j=2}^{n} A_j(x_j^s - x_j^{s+1})$ is 0 it will.

The optimality conditions consists of (2.24) and (2.25) and the dual feasibility condition (2.25) for $i = n$ will always be fulfilled. The residuals for the remaining conditions are r^{s+1} and q_i^{s+1}, $i = 1, ..n-1$, for the primal and dual conditions, respectively. As the ADMM iterations go to infinity the residuals go to zero [6], thus if the residuals are small, then the problem approaches optimality. This means that reasonable stopping criteria are that the residuals are small enough, *i.e.*,

$$\|r^s\|_2 \leq \epsilon^{\text{primal}} \quad \text{and} \quad \|q^s\|_2 \leq \epsilon^{\text{dual}}, \tag{2.29}$$

where $\epsilon^{\text{primal}} > 0$ and $\epsilon^{\text{dual}} > 0$ are the primal and dual tolerances respectively. The tolerances scale with the room in which the residuals lie, meaning $\epsilon^{\text{primal}} = \sqrt{p} \cdot \epsilon$ since the primal residual is in $\mathbb{R}^p$ and $\epsilon^{\text{dual}} = \sqrt{\sum_{i=1}^{n-1} m_i} \cdot \epsilon$ since the dual residual is in $\mathbb{R}^{m_1+...+m_{n-1}}$ [6].

2.6 Subgradient optimization

Subgradient methods are iterative algorithms which can be used to solve optimization problems for both differentiable and non-differentiable functions, for example the Lagrangian dual function.

Since the Lagrangian dual function consists of several linear segments, in an intersection by two or more segments, it is not differentiable. Instead we introduce the subdifferential [2, Chapter 17].

The subdifferential $\partial f(\boldsymbol{x})$ of a concave function $f : \mathbb{R}^m \mapsto \mathbb{R}$ is defined as the set of vectors $\boldsymbol{\gamma} \in \mathbb{R}^m$, which at a given point $\tilde{\boldsymbol{x}} \in \mathbb{R}^m$ forms a supporting hyperplane to the function f for all points $\boldsymbol{x} \in \mathbb{R}^m$

$$\partial f(\tilde{\boldsymbol{x}}) := \left\{ \boldsymbol{\gamma} \in \mathbb{R}^m \;\middle|\; f(\boldsymbol{x}) \leq f(\tilde{\boldsymbol{x}}) + \boldsymbol{\gamma}^\top(\boldsymbol{x} - \tilde{\boldsymbol{x}}),\ \forall \boldsymbol{x} \in \mathbb{R}^m \right\}. \tag{2.30}$$

A simple subgradient method iteratively updates the dual variables to convergence towards the optimal value by taking steps in subgradient directions. To do this, one first initializes the dual variables $\boldsymbol{u}^0$. Then, evaluate the Lagrangian dual function with the current dual variables in order to get a lower bound; see Section 2.2. Next, one calculates a subgradient γ_s to the Lagrangian dual function, to determine the step direction. Next, one determines the step length δ_s, to know how far one shall step in the subgradient direction. Lastly, one updates the dual variables with step length times the subgradient γ_s. This continues until some stopping criteria has been met [2, Chapter 17].

The convergence rate will depend on the choice of step length. For many choices of step lengths, subgradient methods will zigzag its way towards optimum. Zigzagging patterns are correlated with a slow convergence. There are several ways to speed up the convergence. First, one can use *conditional subgradients*. If the current dual

variable $\boldsymbol{u}$ is at the boundary of the feasible region for the dual variables, and if the subgradient is pointing out of the set; the conditional subgradient will project back the subgradient to the set of feasible directions, before calculating the step length. This can yield a larger step length, and increase the speed of convergence.

Another way is to use *modified deflected subgradients* (MDS). This is a combination of *average direction strategy* (ADS) and *modified gradient technique* (MGT). MDS uses a convex combination of the previous step direction and the current subgradient direction [9]. This decreases the zigzagging pattern and can converge more directly towards an optimal solution [9]. This indicates a faster convergence.

To use this method, we first determine the Lagrangian dual function for the problem, i.e., $g(\boldsymbol{u})$, as defined in (2.4). Then we initialize the step direction $\boldsymbol{v}^0$ and the step length α_0.

First, we determine the parameter β_s according to

$$\beta_s := \left[-\frac{(\boldsymbol{\gamma}^s)^\top \boldsymbol{v}^{s-1}}{\|\boldsymbol{\gamma}^s\| \, \|\boldsymbol{v}^{s-1}\|} \right]_+, \tag{2.31}$$

where $\boldsymbol{\gamma}^s$ is the current iteration's subgradient, $\boldsymbol{v}^{s-1}$ is the previous iterations step direction, $||\cdot||$ is the L_2-norm and $[\boldsymbol{x}]_+ := \max\{0, \boldsymbol{x}\}$. Then we determine the parameter $\eta_s \in [0.5, 1]$ as

$$\eta_s := \frac{1}{2 - \beta_s}, \tag{2.32a}$$

as well as the step lengths for ADS, MGT and MDS, according to

$$\Gamma^s_{\text{ADS}} := \frac{\|\boldsymbol{\gamma}^s\|}{\|\boldsymbol{v}^{s-1}\|}, \tag{2.32b}$$

$$\Gamma^s_{\text{MGT}} := \left[-\eta_s \frac{(\boldsymbol{\gamma}^s)^\top \boldsymbol{v}^{s-1}}{\left\|\boldsymbol{v}^{s-1}\right\|^2} \right]_+ \tag{2.32c}$$

and

$$\Gamma^s_{\text{MDS}} := (1 - \beta_s)\Gamma^s_{\text{MGT}} + \beta_s \Gamma^s_{\text{ADS}}, \tag{2.32d}$$

respectively. From this in accordance with MDS, the current iteration's step direction is determined as

$$\boldsymbol{v}^s := \boldsymbol{\gamma}^s + \Gamma^s_{MDS} \boldsymbol{v}^{s-1}, \tag{2.33}$$

as well as the step length according to the Polyak step length rule

$$\delta_s := \frac{\alpha_s \left(\bar{z} - g(\boldsymbol{u}^s) \right)}{\|\boldsymbol{v}^s\|^2}. \tag{2.34}$$

Finally, the dual variables are updated as

$$\boldsymbol{u}^{s+1} := \boldsymbol{u}^s + \delta_s \boldsymbol{v}^s \tag{2.35}$$

and are then projected onto the feasible region.

2.7 An algorithm for solution of the assignment subproblems

Assignment solving algorithms are special solution methods used to solve assignment problems. The main characteristic of a 2D assignment problem is that there are two groups of objects and each object is to be assigned to one object in the other group. However, there are variants of assignment problems with slight differences in how the objects are allowed to be assigned. The 2D assignment problem can be expressed as a matrix, where one group of objects is represented by the rows and the other group by the columns. If a row is connected to a column, then this connection is represented by a 1 in the assignment matrix. If there is no connection the element is 0 in the assignment matrix. One variant is square assignment problems where the number of rows and columns are the same. For this variant, each row and column has to sum to one. Another variant is rectangular assignment problems where the number of rows and columns are not equal, meaning that each object cannot be assigned to exactly one other. For this variant, each row has to sum to one if the number of rows is smaller than the number of columns and vice versa. Some algorithms work for both square and rectangular problems while others only work for one type. The *Jonker-Volgenant-Castanon* (JVC) algorithm is one such algorithm which solves both square and rectangular assignment problems. The JVC algorithm uses graph theory and a smart initialization in order to solve the assignment problem. In theory, the complexity of the JVC algorithm is $\mathcal{O}(n^3)$ but in practice it is much faster [10].

3

Implementation

This section presents in detail how the two methods ADMM and MDS are applied to the given optimization problem (1.7). Code for the two methods, link to all instances used, can be found in Appendix C.

3.1 An initialization heuristic

For the Polyak step length rule, we need an upper bound. The heuristic iterative solves the T assignment problems. In the next iteration s, it changes the cost matrix based on the previous solution to the assignment problems. For every element $d_{k,i,j}$ besides the dummy variables, we add the switching penalty times the value of a function evaluated in the previous iteration $s-1$:

$$f(w_{k,i,j}^{s-1}) = \begin{cases} 1 - w_{k+1,i,j}^{s-1} - 10^{-3}, & \text{if } k = 1, & (i,j) \in \hat{\mathcal{I}} \\ 1 - w_{k-1,i,j}^{s-1} - w_{k+1,i,j}^{s-1} - 10^{-3}, & \text{if } k \in \hat{\mathcal{K}} \setminus \{1\}, & (i,j) \in \hat{\mathcal{I}} \\ 1 - w_{k-1,i,j}^{s-1} - 10^{-3}, & \text{if } k = T, & (i,j) \in \hat{\mathcal{I}} \\ 0, & \forall k \in \mathcal{K}, & (i,j) \notin \hat{\mathcal{I}} \end{cases} \tag{3.1}$$

and then solve

$$W^s := \underset{W}{\operatorname{argmin}} \quad \sum_{k=1}^{T} \sum_{i=1}^{n_{\mathcal{X}}+1} \sum_{j=1}^{n_{\mathcal{Y}}+1} \left(d_{k,i,j} + \frac{\mu^p}{2} f(w_{k,i,j}^{s-1}) \right) w_{k,i,j} \tag{3.2a}$$

$$\text{subject to} \quad W_k \in \overline{\mathcal{W}}, \quad k \in \mathcal{K} \tag{3.2b}$$

The function value is multiplied with the switching penalty $\mu^p/2$, in order to compensate for switching assignments between time steps. A small constant 10^{-3} is subtracted from the function. This makes sure that when solving, it is cheaper to assign a ground truth to something that does exist instead of assigning them both to dummies. This is done in order to avoid getting stuck in suboptimal solutions where too much is assigned to dummies.

As one can see, if the neighboring time steps $w_{k-1,i,j}^{s-1} = 1$ and $w_{k+1,i,j}^{s-1} = 1$ were assigned as such for some i and j in the previous iteration, it makes it cheaper to assign $w_{k,i,j}^{s} = 1$ in the currant iteration.

The heuristic is terminated when the assignment matrix is the same in two adjacent iterations, $W^s = W^{s-1}$, or after T iterations. After termination, the number of track switches is determined. This yields an upper bound $\bar{z}$ on the optimal value $z^\star$, which can be used in Polyak step length. For initialization to ADMM, the output is instead the solution $\bar{W}$ and $\bar{H}$, which can be used to calculate $\bar{\Phi}$ and $\bar{\Psi}$.

3.2 Implementation of the ADMM

Consider the optimization problem (1.7). Since ADMM applies to equality constraints, we rewrite the two inequality constraints into equality constraints. We add a non-negative slack variable to each of the constraints (1.7c) and (1.7d). Also, we add a non-negativity constraint to the help variable $h_{k,i,j}$, such that we get

$$
\begin{aligned}
h_{k,i,j} &= w_{k,i,j} - w_{k+1,i,j} + \phi_{k,i,j}, && k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, && \text{(3.3a)}\\
h_{k,i,j} &= w_{k+1,i,j} - w_{k,i,j} + \psi_{k,i,j}, && k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, && \text{(3.3b)}\\
\phi_{k,i,j} &\geq 0, && k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, && \text{(3.3c)}\\
\psi_{k,i,j} &\geq 0, && k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, && \text{(3.3d)}\\
h_{k,i,j} &\geq 0, && k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}. && \text{(3.3e)}
\end{aligned}
$$

ADMM also requires the feasible set to be convex, therefore the non-convex assignment set $\mathcal{W}$ is LP relaxed to the convex set $\overline{\mathcal{W}}$. Using the definition presented in (2.18), the augmented Lagrangian can be written as

$$
\begin{aligned}
L_\rho(W,H,\Phi,\Psi,\Lambda,\Xi) = &\sum_{k=1}^{T}\sum_{i=1}^{n_\mathcal{X}+1}\sum_{j=1}^{n_\mathcal{Y}+1} d_{k,i,j}w_{k,i,j} + \frac{\mu^p}{2}\sum_{k=1}^{T-1}\sum_{i=1}^{n_\mathcal{X}}\sum_{j=1}^{n_\mathcal{Y}} h_{k,i,j}\\
&+ \sum_{k=1}^{T-1}\sum_{i=1}^{n_\mathcal{X}}\sum_{j=1}^{n_\mathcal{Y}} \lambda_{k,i,j}\Big(w_{k,i,j} - w_{k+1,i,j} - h_{k,i,j} + \phi_{k,i,j}\Big)\\
&+ \sum_{k=1}^{T-1}\sum_{i=1}^{n_\mathcal{X}}\sum_{j=1}^{n_\mathcal{Y}} \xi_{k,i,j}\Big(w_{k+1,i,j} - w_{k,i,j} - h_{k,i,j} + \psi_{k,i,j}\Big)\\
&+ \frac{\rho}{2}\sum_{k=1}^{T-1}\sum_{i=1}^{n_\mathcal{X}}\sum_{j=1}^{n_\mathcal{Y}} \Big(w_{k,i,j} - w_{k+1,i,j} - h_{k,i,j} + \phi_{k,i,j}\Big)^2\\
&+ \frac{\rho}{2}\sum_{k=1}^{T-1}\sum_{i=1}^{n_\mathcal{X}}\sum_{j=1}^{n_\mathcal{Y}} \Big(w_{k+1,i,j} - w_{k,i,j} - h_{k,i,j} + \psi_{k,i,j}\Big)^2 && \text{(3.4a)}\\
&\phi_{k,i,j} \geq 0, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, && \text{(3.4b)}\\
&\psi_{k,i,j} \geq 0, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, && \text{(3.4c)}\\
&h_{k,i,j} \geq 0, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, && \text{(3.4d)}
\end{aligned}
$$

where $\lambda_{k,i,j}$ are free dual variables for constraint (3.3a) and $\xi_{k,i,j}$ are free dual variables for constraint (3.3b). Then, we form the subproblems where each subproblem will be solved sequentially. First, we update the assignment variables in every time

step $k \in \mathcal{K}$ and get the update

$$W_1^{s+1} := \underset{W_1 \in \overline{\mathcal{W}}}{\operatorname{argmin}} L_\rho\Big(W_1, W_2^s, ..., W_T^s, H^s, \Phi^s, \Psi^s, \Lambda^s, \Xi^s\Big) \tag{3.5a}$$

$$W_k^{s+1} := \underset{W_k \in \overline{\mathcal{W}}}{\operatorname{argmin}} L_\rho\Big(W_1^{s+1}, .., W_{k-1}^{s+1}, W_k, W_{k+1}^s, ..., W_T^s, H^s, \Phi^s, \Psi^s, \Lambda^s, \Xi^s\Big), \quad k \in \hat{\mathcal{K}} \setminus \{1\} \tag{3.5b}$$

$$W_T^{s+1} := \underset{W_T \in \overline{\mathcal{W}}}{\operatorname{argmin}} L_\rho\Big(W_1^{s+1}, ..., W_{T-1}^{s+1}, W_T, H^s, \Phi^s, \Psi^s, \Lambda^s, \Xi^s\Big). \tag{3.5c}$$

The updates involve solving T assignment problems with some quadratic objective term, more specifically the subproblem for updating W_k looks like

$$\begin{aligned} L_\rho\Big(W_1^{s+1}, ..., W_{k-1}^{s+1}, W_k, W_{k+1}^s, ..., W_T^s, H^s, \Phi^s, \Psi^s, \Lambda^s, \Xi^s\Big) = \\ \sum_{i=1}^{n_\mathcal{X}+1} \sum_{j=1}^{n_\mathcal{Y}+1} d_{k,i,j} w_{k,i,j} + \sum_{i=1}^{n_\mathcal{X}} \sum_{j=1}^{n_\mathcal{Y}} \Bigg(\Big(-\lambda_{k-1,i,j}^s + \lambda_{k,i,j}^s + \xi_{k-1,i,j}^s \\ - \xi_{k,i,j}^s - 2\rho \cdot w_{k-1,i,j}^{s+1} - 2\rho \cdot w_{k+1,i,j}^s - \rho \cdot \phi_{k-1,i,j}^s + \rho \cdot \phi_{k,i,j}^s \\ + \rho \cdot \psi_{k-1,i,j}^s - \rho \cdot \psi_{k,i,j}^s \Big) w_{k,i,j} + 2\rho \cdot w_{k,i,j}^2 \Bigg) + C_k^W, \qquad k \in \hat{\mathcal{K}} \setminus \{1\}, \end{aligned} \tag{3.6}$$

where C_k^W is some constant that does not depend on W_k. The problems for W_1 and W_T look the same, but without some of the constants before the linear term. They are all quadratic which means that the update steps involve solving an assignment problem with a quadratic objective function, which are hard to solve. Instead we use the theory in Section 2.4 and since the quadratic term can be rewritten as $W_k^\top Q W_k$ where $Q = 2\rho I$, where I is the identity matrix. Since W_k is binary and Q is diagonal (2.16) holds and the quadratic and linear problems are equivalent. The update for

the assignment variables W can be written as

$$W_1^{s+1} := \underset{W_1 \in \overline{\mathcal{W}}}{\operatorname{argmin}} \left\{ \sum_{i=1}^{n_{\mathcal{X}}+1} \sum_{j=1}^{n_{\mathcal{Y}}+1} d_{1,i,j} w_{1,i,j} + \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \left(\Big(\lambda_{1,i,j}^s - \xi_{1,i,j}^s \right. \right.$$
$$\left. \left. -2\rho \cdot w_{2,i,j}^s + \rho \cdot \phi_{1,i,j}^s - \rho \cdot \psi_{1,i,j}^s + 2\rho \Big) w_{1,i,j} \right) + C_1^W \right\} \tag{3.7a}$$

$$W_k^{s+1} := \underset{W_k \in \overline{\mathcal{W}}}{\operatorname{argmin}} \left\{ \sum_{i=1}^{n_{\mathcal{X}}+1} \sum_{j=1}^{n_{\mathcal{Y}}+1} d_{k,i,j} w_{k,i,j} + \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \left(\Big(-\lambda_{k-1,i,j}^s + \lambda_{k,i,j}^s + \xi_{k-1,i,j}^s \right. \right.$$
$$- \xi_{k,i,j}^s - 2\rho \cdot w_{k-1,i,j}^{s+1} - 2\rho \cdot w_{k+1,i,j}^s - \rho \cdot \phi_{k-1,i,j}^s + \rho \cdot \phi_{k,i,j}^s$$
$$\left. \left. +\rho \cdot \psi_{k-1,i,j}^s - \rho \cdot \psi_{k,i,j}^s + 2\rho \Big) w_{k,i,j} \right) + C_k^W \right\}, \quad k \in \hat{\mathcal{K}} \setminus \{1\} \tag{3.7b}$$

$$W_T^{s+1} := \underset{W_T \in \overline{\mathcal{W}}}{\operatorname{argmin}} \left\{ \sum_{i=1}^{n_{\mathcal{X}}+1} \sum_{j=1}^{n_{\mathcal{Y}}+1} d_{T,i,j} w_{T,i,j} + \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \left(\Big(-\lambda_{T-1,i,j}^s + \xi_{T-1,i,j}^s \right. \right.$$
$$\left. \left. -2\rho \cdot w_{T-1,i,j}^{s+1} + \rho \cdot \psi_{T-1,i,j}^s - \rho \cdot \phi_{T-1,i,j}^s + 2\rho \Big) w_{T,i,j} \right) + C_T^W \right\}. \tag{3.7c}$$

The updates for H, Φ, and Ψ are all very similar, they involve minimizing a convex quadratic function over some convex set. Their update rule looks like

$$H^{s+1} := \underset{H \geq 0}{\operatorname{argmin}}\, L_\rho(W^{s+1}, H, \Phi^s, \Psi^s, \Lambda^s, \Xi^s) \tag{3.8a}$$
$$\Phi^{s+1} := \underset{\Phi \geq 0}{\operatorname{argmin}}\, L_\rho(W^{s+1}, H^{s+1}, \Phi, \Psi^s, \Lambda^s, \Xi^s) \tag{3.8b}$$
$$\Psi^{s+1} := \underset{\Psi \geq 0}{\operatorname{argmin}}\, L_\rho(W^{s+1}, H^{s+1}, \Phi^{s+1}, \Psi, \Lambda^s, \Xi^s) \tag{3.8c}$$

and more specifically, for H we have that

$$L_\rho\Big(W^{s+1}, H_1^{s+1}, ..., H_{k-1}^{s+1}, H_k, H_{k+1}^s, ..., H_{T-1}^s, \Phi^s, \Psi^s, \Lambda^s, \Xi^s\Big) =$$
$$\sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \left(\Big(\frac{\mu^p}{2} - \lambda_{k,i,j}^s - \xi_{k,i,j}^s - \rho \cdot \phi_{k,i,j}^s - \rho \cdot \psi_{k,i,j}^s \Big) h_{k,i,j} \right.$$
$$\left. + \rho \cdot h_{k,i,j}^2 \right) + C_k^H, \quad k \in \hat{\mathcal{K}} \setminus \{1\}, \tag{3.9}$$

which means that the update rule for H is separable with respect to k, i and j. This means that the update of H is done by finding the minimum to a quadratic function with the constraint that the variable has to be positive. The update then becomes

the derivative of the function (3.9) projected onto the positive real line as

$$h_{k,i,j}^{s+1} = \left[\frac{\mu^p}{4\rho} - \frac{1}{2\rho}(\lambda_{k,i,j}^s + \xi_{k,i,j}^s) - \frac{1}{2}(\phi_{k,i,j}^s + \psi_{k,i,j}^s)\right]_+, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}} \tag{3.10a}$$

$$\phi_{k,i,j}^{s+1} = \left[\frac{1}{\rho}\lambda_{k,i,j}^s + w_{k,i,j}^{s+1} - w_{k+1,i,j}^{s+1} - h_{k,i,j}^s\right]_+, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}} \tag{3.10b}$$

$$\psi_{k,i,j}^{s+1} = \left[\frac{1}{\rho}\xi_{k,i,j}^s - w_{k,i,j}^{s+1} + w_{k+1,i,j}^{s+1} - h_{k,i,j}^s\right]_+, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}. \tag{3.10c}$$

Finally, the dual variables are updated, their update rule is given in (2.19d) and when it is applied to our problem it gives

$$\lambda_{k,i,j}^{s+1} := \lambda_{k,i,j}^s + \rho(w_{k,i,j}^{s+1} - w_{k+1,i,j}^{s+1} - h_{k,i,j}^{s+1} + \phi_{k,i,j}^{s+1}), \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, \tag{3.11a}$$

$$\xi_{k,i,j}^{s+1} := \xi_{k,i,j}^s + \rho(w_{k+1,i,j}^{s+1} - w_{k,i,j}^{s+1} - h_{k,i,j}^{s+1} + \psi_{k,i,j}^{s+1}), \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}. \tag{3.11b}$$

The whole algorithm can then be summarized as Algorithm 1 where the update

Algorithm 1: ADMM

Data: $\mathcal{X}, \mathcal{Y}$
Result: $z^\star$
Initializing using heuristic, see Section 3.1
while *Termination criteria not reached* **do**
    Update W
    Update H
    Update Φ
    Update Ψ
    Update Λ
    Update Ξ
end

rules are as stated previously. The only things left are to formulate the termination criteria and verify convergence. As stated in Section 2.5 there are two termination criteria that have to be fulfilled in order to allow for termination. The termination criteria state that the problem has to be both primal and dual feasible, meaning that both the primal and dual residuals have to be small enough. The primal feasibility states that the solution has to be feasible in the primal problem, i.e., the problem before the constraints were relaxed. The primal feasibility conditions look like

$$0 = w_{k,i,j}^\star - w_{k+1,i,j}^\star - h_{k,i,j}^\star + \phi_{k,i,j}^\star, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}} \tag{3.12a}$$

$$0 = w_{k+1,i,j}^\star - w_{k,i,j}^\star - h_{k,i,j}^\star + \psi_{k,i,j}^\star, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}. \tag{3.12b}$$

We can then introduce the primal residual r and let it be defined by the right-hand side of these two equations. More specifically it will be defined as

$$r_{1,k,i,j}^s := w_{k,i,j}^s - w_{k+1,i,j}^s - h_{k,i,j}^s + \phi_{k,i,j}^s \tag{3.13a}$$

$$r_{2,k,i,j}^s := w_{k+1,i,j}^s - w_{k,i,j}^s - h_{k,i,j}^s + \psi_{k,i,j}^s \tag{3.13b}$$

for $k \in \hat{\mathcal{K}}$, $(i,j) \in \hat{\mathcal{I}}$ and the termination criteria for the primal feasibility can then be formulated as $\| r^s \| < \epsilon^{\text{primal}}$. Here ϵ^{primal} is the primal tolerance and it is defined as $\epsilon^{\text{primal}} := \epsilon^{\text{abs}} \cdot \sqrt{2 \cdot (T-1) \cdot n_{\mathcal{X}} \cdot n_{\mathcal{Y}}}$ where ϵ^{abs} is some absolute tolerance and the square root is to compensate for the dimension that the residual resides in.

The dual feasibility conditions look like

$$0 \in \partial\Big(d_{1,i,j} w^\star_{1,i,j}\Big) + \lambda^\star_{1,i,j} - \xi^\star_{1,i,j}, \qquad (i,j) \in \hat{\mathcal{I}} \tag{3.14a}$$

$$0 \in \partial\Big(d_{k,i,j} w^\star_{k,i,j}\Big) + \lambda^\star_{k,i,j} - \xi^\star_{k,i,j} - \lambda^\star_{k-1,i,j} + \xi^\star_{k-1,i,j}, \quad k \in \hat{\mathcal{K}} \setminus \{1\}, \quad (i,j) \in \hat{\mathcal{I}} \tag{3.14b}$$

$$0 \in \partial\Big(d_{T,i,j} w^\star_{T,i,j}\Big) - \lambda^\star_{T-1,i,j} + \xi^\star_{T-1,i,j}, \qquad (i,j) \in \hat{\mathcal{I}} \tag{3.14c}$$

$$0 \in \partial\Big(\frac{\mu^p}{2} h^\star_{k,i,j}\Big) - \lambda^\star_{k,i,j} - \xi^\star_{k,i,j}, \qquad k \in \hat{\mathcal{K}}, \qquad (i,j) \in \hat{\mathcal{I}} \tag{3.14d}$$

$$0 \in \partial\Big(0 \cdot \phi^\star_{k,i,j}\Big) + \lambda^\star_{k,i,j}, \qquad k \in \hat{\mathcal{K}}, \qquad (i,j) \in \hat{\mathcal{I}} \tag{3.14e}$$

$$0 \in \partial\Big(0 \cdot \psi^\star_{k,i,j}\Big) + \xi^\star_{k,i,j}, \qquad k \in \hat{\mathcal{K}}, \qquad (i,j) \in \hat{\mathcal{I}}. \tag{3.14f}$$

We have by definition that Ψ^{s+1} minimizes $L_\rho(W^{s+1}, H^{s+1}, \Phi^{s+1}, \Psi, \Lambda^s, \Xi^s)$ and therefore

$$\begin{aligned} 0 &\in \partial\Big(0 \cdot \psi^{s+1}_{k,i,j}\Big) + \xi^s_{k,i,j} + \rho\Big(w^{s+1}_{k+1,i,j} - w^{s+1}_{k,i,j} - h^{s+1}_{k,i,j} + \psi^{s+1}_{k,i,j}\Big) \\ &= \partial\Big(0 \cdot \psi^{s+1}_{k,i,j}\Big) + \xi^s_{k,i,j} + \rho r^{s+1}_{2,k,i,j} \\ &= \partial\Big(0 \cdot \psi^{s+1}_{k,i,j}\Big) + \xi^{s+1}_{k,i,j}, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}. \end{aligned}$$

This means that Ψ^{s+1} and Ξ^{s+1} always satisfy (3.14f); the same calculations can be done for Φ^{s+1} and (3.14e) resulting in the same conclusion for Φ^{s+1} and Λ^{s+1}, which can be seen in Appendix A. Then we continue with H^{s+1} and (3.14d) and get

$$\begin{aligned} 0 \in &\partial\Big(\frac{\mu^p}{2} h^{s+1}_{k,i,j}\Big) - \lambda^s_{k,i,j} - \xi^s_{k,i,j} - \rho\Big(w^{s+1}_{k,i,j} - w^{s+1}_{k+1,i,j} - h^{s+1}_{k,i,j} + \phi^s_{k,i,j}\Big) \\ &- \rho\Big(w^{s+1}_{k+1,i,j} - w^{s+1}_{k,i,j} - h^{s+1}_{k,i,j} + \psi^s_{k,i,j}\Big) \\ = &\partial\Big(\frac{\mu^p}{2} h^{s+1}_{k,i,j}\Big) - \lambda^{s+1}_{k,i,j} - \xi^{s+1}_{k,i,j} \\ &- \rho\Big(-\phi^{s+1}_{k,i,j} + \phi^s_{k,i,j} - \psi^{s+1}_{k,i,j} + \psi^s_{k,i,j}\Big), \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}. \end{aligned}$$

This is not always fulfilled, however it is fulfilled if $\rho(-\phi^{s+1}_{k,i,j}+\phi^{s}_{k,i,j}-\psi^{s+1}_{k,i,j}+\psi^{s}_{k,i,j}) \in \partial\Big(\frac{\mu^p}{2}h^{s+1}_{k,i,j}\Big) - \lambda^{s+1}_{k,i,j} - \xi^{s+1}_{k,i,j}$ for $k \in \hat{\mathcal{K}}$, $(i,j) \in \hat{\mathcal{I}}$. We get similar expressions for W and in the end, we have

$$\rho\Big(-2w^{s+1}_{2,i,j}+2w^{s}_{2,i,j}+\phi^{s+1}_{1,i,j}-\phi^{s}_{1,i,j}-\psi^{s+1}_{1,i,j}+\psi^{s}_{1,i,j}\Big) \in \partial\Big(d_{1,i,j}w^{s+1}_{1,i,j}\Big) + \lambda^{s+1}_{1,i,j} - \xi^{s+1}_{1,i,j}, \quad (i,j) \in \hat{\mathcal{I}} \tag{3.15a}$$

$$\rho\Big(-2w^{s+1}_{k+1,i,j}+2w^{s}_{k+1,i,j}-\phi^{s+1}_{k-1,i,j}+\phi^{s}_{k-1,i,j}+\psi^{s+1}_{k-1,i,j}-\psi^{s}_{k-1,i,j} + \phi^{s+1}_{k,i,j}-\phi^{s}_{k,i,j}-\psi^{s+1}_{k,i,j}+\psi^{s}_{k,i,j}\Big) \in \partial\Big(d_{k,i,j}w^{s+1}_{k,i,j}\Big) + \lambda^{s+1}_{k,i,j} - \xi^{s+1}_{k,i,j} - \lambda^{s+1}_{k-1,i,j} + \xi^{s+1}_{k-1,i,j}, \quad k \in \hat{\mathcal{K}} \setminus \{1\}, \quad (i,j) \in \hat{\mathcal{I}} \tag{3.15b}$$

$$\rho\Big(-\phi^{s+1}_{T-1,i,j}+\phi^{s}_{T-1,i,j}+\psi^{s+1}_{T-1,i,j}-\psi^{s}_{T-1,i,j}\Big) \in \partial\Big(d_{T,i,j}w^{s+1}_{T,i,j}\Big) - \lambda^{s+1}_{T-1,i,j} + \xi^{s+1}_{T-1,i,j}, \quad (i,j) \in \hat{\mathcal{I}} \tag{3.15c}$$

$$\rho\Big(-\phi^{s+1}_{k,i,j}+\phi^{s}_{k,i,j}-\psi^{s+1}_{k,i,j}+\psi^{s}_{k,i,j}\Big) \in \partial\Big(\frac{\mu^p}{2}h^{s+1}_{k,i,j}\Big) - \lambda^{s+1}_{k,i,j} - \xi^{s+1}_{k,i,j}, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, \tag{3.15d}$$

for $W_1^{s+1}, W_k^{s+1}, W_T^{s+1}$, and H^{s+1}, respectively; the calculations for the rest can be seen in Appendix A. From these equations, we wish to construct the dual residual q, instead of letting all these expressions go to zero we can let their parts go to zero. We note that all formulas in (3.15) include some difference of the variables W, Φ, and Ψ; therefore instead of letting (3.15) go to zero, we can let their parts go to zero. We construct q_1, q_2, and q_3 to be the difference between two iteration steps s and $s+1$ for W, Φ, and Ψ, like

$$q^{s+1}_{1,k,i,j} := \begin{cases} w^{s+1}_{k,i,j} - w^{s}_{k,i,j}, & \text{if} \quad k \in \mathcal{K} \setminus \{1\}, \quad (i,j) \in \hat{\mathcal{I}}, \\ 0, & \text{if} \quad k = 1, \quad (i,j) \in \hat{\mathcal{I}}, \end{cases} \tag{3.16a}$$

$$q^{s+1}_{2,k,i,j} := \begin{cases} \phi^{s+1}_{k,i,j} - \phi^{s}_{k,i,j}, & \text{if} \quad k \in \mathcal{K} \setminus \{T\}, \quad (i,j) \in \hat{\mathcal{I}}, \\ 0, & \text{if} \quad k = T, \quad (i,j) \in \hat{\mathcal{I}}, \end{cases} \tag{3.16b}$$

$$q^{s+1}_{3,k,i,j} := \begin{cases} \psi^{s+1}_{k,i,j} - \psi^{s}_{k,i,j}, & \text{if} \quad k \in \mathcal{K} \setminus \{T\}, \quad (i,j) \in \hat{\mathcal{I}}, \\ 0, & \text{if} \quad k = T, \quad (i,j) \in \hat{\mathcal{I}}. \end{cases} \tag{3.16c}$$

Let $Q = (q_{1,k,i,j}, q_{2,k,i,j}, q_{3,k,i,j} \mid k \in \mathcal{K},\ (i,j) \in \hat{\mathcal{I}})$. The termination criteria for the dual feasibility can then be formulated in the same way as for the primal, as $\| Q^s \| < \epsilon^{\text{dual}}$. In the same way as for the primal, ϵ^{dual} is the dual tolerance and it is defined as $\epsilon^{\text{dual}} := \epsilon^{\text{abs}} \cdot \sqrt{3 \cdot (T-1) \cdot n_{\mathcal{X}} \cdot n_{\mathcal{Y}}}$. Finally, we will check the

termination criteria. Assumptions 2.5.1 and 2.5.2 state that the functions that make up the objective function are closed, proper, and convex and that the unaugmented Lagrangian has a saddle point. The objective function (1.7) is a sum of other functions which are separable with respect to the variables. The functions are all linear which means that they are all convex. A proper convex function has a non-empty domain, is never $-\infty$, and has at least one point that is not ∞. All functions are linear, all variables are larger than or equal to zero and the slope of the functions is positive since $d_{k,i,j} \geq 0$ and $\mu^p/2 \geq 0$. This means that the functions never take the value $-\infty$. A point which has a function value that is not ∞, can be, for example, by assigning all ground truths $i \in \{1, ...n_{\mathcal{X}}\}$ to dummy estimate $n_{\mathcal{Y}} + 1$, and all estimates $j \in \{1, ..., n_{\mathcal{Y}}\}$ to dummy ground truth $n_{\mathcal{X}} + 1$. Since this is interpreted as leaving everything unsigned, there will be no track switches - therefore switching cost is 0. Also, since $0 \leq d_{k,i,j} \leq c^p/2$, it will generate a finite function value for our case, as long as $c^p < \infty$. As a result, we know that our functions are proper and convex which means we can use the fact that closed and lower semi-continuous are analogous to proper convex functions [11]. We know the functions are linear which means they are continuous which is stronger than lower semi-continuous, therefore the functions are closed, proper, and convex.

In order to prove we have a saddle point, we use the definition, that if the Hessian matrix to the function is indefinite at some point then the function has a saddle point at that point [12]. The Hessian is a matrix of the second-order derivatives of the function, it is symmetric by definition. In our case the unaugmented Lagrangian is

$$\begin{aligned} L_0(W, H, \Phi, \Psi, \Lambda, \Xi) = &\sum_{i=1}^{n_{\mathcal{X}}+1} \sum_{j=1}^{n_{\mathcal{Y}}+1} d_{k,i,j} w_{k,i,j} + \frac{\mu^p}{2} \sum_{k=1}^{T-1} \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} h_{k,i,j} \\ &+ \sum_{k=1}^{T-1} \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \lambda_{k,i,j} \Big(w_{k,i,j} - w_{k+1,i,j} - h_{k,i,j} + \phi_{k,i,j} \Big) \\ &+ \sum_{k=1}^{T-1} \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \xi_{k,i,j} \Big(w_{k+1,i,j} - w_{k,i,j} - h_{k,i,j} + \psi_{k,i,j} \Big) \end{aligned} \tag{3.17}$$

which means that the Hessian for our function will be symmetric with a zero diagonal since our function is linear with respect to any one variable. In order to show that the Hessian is indefinite we wish to calculate the eigenvalues since, by definition, a matrix that is indefinite has both positive and negative eigenvalues [12]. We also know that the trace which is the sum of the diagonal of a matrix equals the sum of the eigenvalues [12]. The diagonal is zero which means that the sum of eigenvalues is zero. Unless every eigenvalue is zero, there exist some eigenvalues that are positive and some that are negative such that they sum to zero. We also know from Cholesky decomposition that a positive semidefinite matrix can be written as $U^\intercal U$, where U is an upper triangular matrix where the elements on the diagonal are allowed to be zero [13]. If we assume this holds for our case then we have no negative eigenvalues and thus all eigenvalues are zero. However if it doesn't hold then at least one eigenvalue is negative and since all eigenvalues sum to zero, at least one has to be positive. We assume it holds and try to construct some matrix U such that $U^\intercal U$ equals our

Hessian. We know that our Hessian has a zero diagonal and the diagonal of $U^\intercal U$ is the inner product of U with itself, meaning U has to be the zero matrix for it to hold, which means that the Hessian also has to be zero which it is not. It therefore holds that the Hessian is not positive semidefinite, and therefore along with the fact that the eigenvalues sum up to zero we can prove that the Hessian has positive and negative eigenvalues and therefore that the unaugmented Lagrangian has a saddle point.

This means that if there exist two sets such that any two matrices in the same set are orthogonal then the multi-block ADMM converges, see Section 2.5. The matrices come from (3.3a) and (3.3b) which can be represented as matrices. It is easy to realize that the vectors representing $w_{k,i,j}$ and $w_{k+1,i,j}$ are not orthogonal. This means that these two cannot be in the same set. It is also easy to realize that the vector for $\phi_{k,i,j}$ is not orthogonal with either $w_{k,i,j}$ or $w_{k+1,i,j}$, which means it cannot be in the same set as either one of them. This means that the sufficient condition does not hold and we cannot guarantee that multi-block ADMM converges.

It is possible that there exists some range for ζ such that our problem converges for the variant of multi-block ADMM. This is however nothing we have confirmed.

3.2.1 ADMM initialization and parameters

ADMM is initialized using the heuristic defined in Section 3.1, which determine the values for W and H. From this, we can calculate Φ using (3.3a) and Ψ using (3.3b).

ADMM is run with varying values of the parameter ρ; it starts with $\rho = 20$ and is multiplied with a factor of 2 if the dual and primal residuals are more than a factor 10 apart. If the dual residual is more than a factor 10 larger than the primal residual then ρ is divided by 2. If the primal residual is more than 10 times as large as the dual residual, then ρ is multiplied by 2. We introduce the parameter ζ in the dual update step and let $\zeta = 1$ for all runs. The termination criteria is that both the primal and dual residuals have to be small enough; the absolute tolerance for the two of them is set to $\epsilon^{\text{abs}} = 10^{-4}$.

3.2.2 ADMM bridges

The main drawback with ADMM is that it is slow to converge, however, there are ways to speed up the convergence. For example, letting ρ vary between iterations or using a smart initialization in order to start at something close to the optimum. We also tried another implementation of ADMM where we introduced redundant constraints and then relaxed them. We saw that ADMM converged slowly and our hypothesis was that since the relaxed constraints only connect to neighboring time steps, it results in a slow flow of information. In order to speed up the flow of information, we introduce some new constraints where time steps further away are connected with each other. This can be done by combining two constraints in (3.3a),

for $k = l$ and $k = l + 1$, namely

$$h_{l,i,j} = w_{l,i,j} - w_{l+1,i,j} + \phi_{l,i,j}, \qquad (i,j) \in \hat{\mathcal{I}}, \tag{3.18a}$$

$$h_{l+1,i,j} = w_{l+1,i,j} - w_{l+2,i,j} + \phi_{l+1,i,j}, \qquad (i,j) \in \hat{\mathcal{I}}. \tag{3.18b}$$

By adding these two equations we find that the $w_{l+1,i,j}$ terms cancel each other while $w_{l,i,j}$ and $w_{l+2,i,j}$ remain. This is known as *telescoping* [14] and can be done for any number of time steps

$$0 = w_{l,i,j} - w_{l+m,i,j} + \sum_{n=0}^{m-1} \Big(\phi_{l+n,i,j} - h_{l+n,i,j} \Big), \quad (i,j) \in \hat{\mathcal{I}}, \tag{3.19}$$

and results in a direct connection between time step l and $m + l$, we call these constraints *bridges*. The downside of this is that all the added constraints have to be relaxed and this introduces a new dual variable for each such constraint.

3.3 A modified deflected subgradient method

Consider the optimization problem (1.7). We Lagrangian relax the two constraints (1.7c) and (1.7d) and multiply them with the non-negative dual variables, $\lambda_{k,i,j}$ and $\xi_{k,i,j}$, respectively, to get the Lagrangian function

$$\begin{aligned}
\mathcal{L}(W,H,\Lambda,\Xi) = &\sum_{k=1}^{T} \sum_{i=1}^{n_{\mathcal{X}}+1} \sum_{j=1}^{n_{\mathcal{Y}}+1} d_{k,i,j} w_{k,i,j} + \frac{\mu^p}{2} \sum_{k=1}^{T-1} \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} h_{k,i,j} \\
&+ \sum_{k=1}^{T-1} \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \lambda_{k,i,j} \Big(w_{k,i,j} - w_{k+1,i,j} - h_{k,i,j} \Big) \\
&+ \sum_{k=1}^{T-1} \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \xi_{k,i,j} \Big(w_{k+1,i,j} - w_{k,i,j} - h_{k,i,j} \Big)
\end{aligned} \tag{3.20}$$

where $W \in \overline{\mathcal{W}}$ and $\Lambda, \Xi \geq 0$. Rearranging the expression above, the Lagrangian function can be written as

$$\begin{aligned}
\mathcal{L}(W,H,\Lambda,\Xi) = &\sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \Big(d_{1,i,j} + \lambda_{1,i,j} - \xi_{1,i,j} \Big) w_{1,i,j} \\
&+ \sum_{k=2}^{T-1} \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \Big(d_{k,i,j} + \lambda_{k,i,j} - \xi_{k,i,j} - \lambda_{k-1,i,j} + \xi_{k-1,i,j} \Big) w_{k,i,j} \\
&+ \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \Big(d_{T,i,j} - \lambda_{T-1,i,j} + \xi_{T-1,i,j} \Big) w_{T,i,j} \\
&+ \sum_{k=1}^{T-1} \sum_{i=1}^{n_{\mathcal{X}}} \sum_{j=1}^{n_{\mathcal{Y}}} \Big(\frac{\mu^p}{2} - \lambda_{k,i,j} - \xi_{k,i,j} \Big) h_{k,i,j} \\
&+ \sum_{k=1}^{T} \left(\sum_{i=1}^{n_{\mathcal{X}}+1} d_{k,i,n_{\mathcal{Y}}+1} w_{k,i,n_{\mathcal{Y}}+1} + \sum_{j=1}^{n_{\mathcal{Y}}+1} d_{k,n_{\mathcal{X}}+1,j} w_{k,n_{\mathcal{X}}+1,j} \right).
\end{aligned} \tag{3.21}$$

Since $h_{k,i,j}$ is a free variable in the Lagrangian function, we can conclude that in order for the dual problem to be optimal

$$\frac{\mu^p}{2} - \lambda_{k,i,j} - \xi_{k,i,j} = 0, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}} \tag{3.22}$$

must hold or else, $h_{k,i,j}$ will take values $\pm\infty$, which cannot be optimal in a bounded solution. From this expression, we can simplify the Lagrangian function into one set of dual variables, where we know that

$$0 \leq \lambda_{k,i,j} \leq \frac{\mu^p}{2}, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}. \tag{3.23}$$

Using (3.22), the Lagrangian dual function can then be simplified into

$$\begin{aligned} g(\Lambda) = \underset{W \in \overline{\mathcal{W}}}{\text{minimium}} \; \Bigg\{ & \sum_{i=1}^{n_\mathcal{X}} \sum_{j=1}^{n_\mathcal{Y}} \Big(d_{1,i,j} + 2 \cdot \lambda_{1,i,j} - \frac{\mu^p}{2} \Big) w_{1,i,j} \\ & + \sum_{k=2}^{T-1} \sum_{i=1}^{n_\mathcal{X}} \sum_{j=1}^{n_\mathcal{Y}} \Big(d_{k,i,j} + 2 \cdot \lambda_{k,i,j} - 2 \cdot \lambda_{k-1,i,j} \Big) w_{k,i,j} \\ & + \sum_{i=1}^{n_\mathcal{X}} \sum_{j=1}^{n_\mathcal{Y}} \Big(d_{T,i,j} - 2 \cdot \lambda_{T-1,i,j} + \frac{\mu^p}{2} \Big) w_{T,i,j} \\ & + \sum_{k=1}^{T} \Bigg(\sum_{i=1}^{n_\mathcal{X}+1} d_{k,i,n_\mathcal{Y}+1} w_{k,i,n_\mathcal{Y}+1} + \sum_{j=1}^{n_\mathcal{Y}+1} d_{k,n_\mathcal{X}+1,j} w_{k,n_\mathcal{X}+1,j} \Bigg) \Bigg\} \end{aligned} \tag{3.24}$$

where the dual function can be separated into T assignment subproblems. We get the Lagrangian dual problem

$$g^\star := \underset{0 \leq \lambda_{k,i,j} \leq \frac{\mu^p}{2}}{\text{maximize}} \Big\{ g(\Lambda) \Big\}. \tag{3.25}$$

In order to maximize the Lagrangian dual function g, we utilize the modified deflected subgradient. First, we initialize the dual variables $\lambda_{k,i,j}$ to $\mu^p/4$, which is equivalent to solving each assignment problem k, without any switching penalty. Then, we solve T assignment subproblems. We calculate a subgradient, $\boldsymbol{\gamma}$, given the solutions to the subproblems, as

$$\gamma_{k,i,j} = w_{k+1,i,j} - w_{k,i,j}, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}, \tag{3.26}$$

where each element in the subgradient to the simplified Lagrangian dual function will be the difference between assignment in the next time step $k+1$ and current time step k.

Then, we use conditional subgradients [15], where the step direction is projected onto the tangent cone of the feasible region at the current point $\lambda_{k,i,j}$, such that any small enough positive step will lead to a feasible point. If $\lambda_{k,i,j} = 0$ and $\gamma_{k,i,j} < 0$, then $\gamma_{k,i,j}$ is projected like $\gamma_{k,i,j} = 0$. If $\lambda_{k,i,j} = \frac{\mu^p}{2}$ and $\gamma_{k,i,j} > 0$, then similarly $\gamma_{k,i,j}$ is projected onto a feasible direction, like $\gamma_{k,i,j} = 0$. Then, we repeat the projection

for step direction $\boldsymbol{v}$. Next, we update the step direction $\boldsymbol{v}$, this is done using (2.31) – (2.33). Then, we determine the step length according to Polyak step length rule; see (2.34).

Finally, we update our dual variables according to (2.35) and project back to the feasible region, an operation which is similar to the computation of the conditional subgradient direction. Namely, if $\lambda_{k,i,j} < 0$ we set $\lambda_{k,i,j} = 0$, and if $\lambda_{k,i,j} > \frac{\mu^p}{2}$, we set $\lambda_{k,i,j} = \frac{\mu^p}{2}$.

Algorithm 2: Modified deflected subgradient

Data: $\mathcal{X}, \mathcal{Y}$
Result: $\underline{z}$
Calculate D, $\bar{z}$
Initializing Λ, $\boldsymbol{v}$, α
while *Termination criteria not reached* **do**
  Update W
  Calculate $\mathcal{L}(W, \Lambda)$, $\boldsymbol{\gamma}(W)$
  if $\mathcal{L}(W, \Lambda) > \underline{z}$ **then**
    $\underline{z} \leftarrow \mathcal{L}(W, \Lambda)$
  end
  Calculate β, η, Γ_{ADS}, Γ_{MGT}, Γ_{MDS}
  if *Updating criteria fullfilled* **then**
    Update α
  end
  Calculate $\boldsymbol{v}$, δ
  Update Λ
end

3.3.1 Subgradient initialization and parameters

The upper bound is calculated ones using the heuristic defined in Section 3.1. Dual variables are initialized to $\mu^p/4$, which is equivalent to solving each assignment problem T, without any switching penalty. The Subgradient method is run with an initial $\alpha = 0.7$ which is decreased by a factor of 0.5 every 20 iterations if there is no improvement to the lower bound. The algorithm has three termination criteria. Firstly, if α becomes smaller than 10^{-3}. Secondly, if the absolute difference between the upper and lower bound becomes smaller than 0.5. Thirdly, if the percentual increase of the lower bound is smaller than 10^{-6} in three consecutive updates, then it terminates. The algorithm also terminates if the subgradient is zero, meaning the optimum is reached by definition.

4

Data, tests, and results

The result covers eighteen + twenty-one instances. The eighteen instances are used for all results except for the scalability where the twenty-one instances are used. The twenty-one instances are only used in order to compare the time complexity between the algorithms. Four of the eighteen instances have been selected to be presented with results more closely. These four instances have been selected due to different pros and cons when it comes to difficulties, to show different types of instances. The instances are named $n_{\mathcal{X}}\_n_{\mathcal{Y}}\_T$, where the first number represents the number of ground truths, the second number represents the number of estimates and the last number represents the number of time steps.

For scalability, seven different numbers, T, of time steps were selected. For each value of T, three different numbers of maximum number of ground truths and estimates were generated. The resulting twenty-one instances were only used for scalability results (see Fig. 4.8).

The data was generated using code from Signal processing. The set $\mathcal{X}$ was generated randomly; where some parameters such as number of time steps T, the birth, and death rate of the trajectories can be specified by the user. The set $\mathcal{Y}$ is then generated by running a MOT algorithm on the randomly generated set $\mathcal{X}$. All data was generated with $c = 20$, $p = 1$, and $\mu = 2$. These are user-specific parameters and determine how much switches are penalized and the maximum distance between ground truth and estimate, as well as how the penalty changes with the distance.

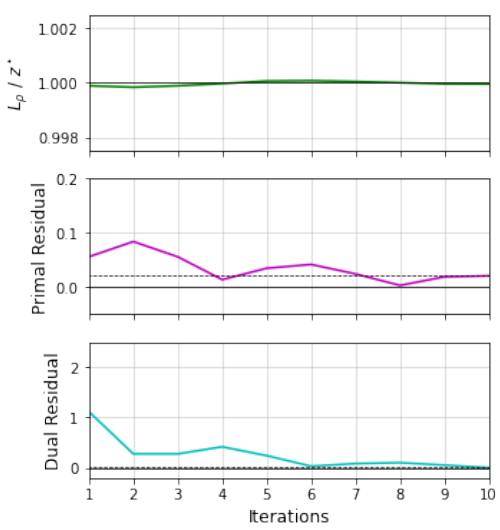
Results are presented with the selected hyperparameters presented in Section 3.2.1 for ADMM and Section 3.3.1 for subgradient, unless other hyperparameters are explicitly specified.

4.1 Results from the ADMM

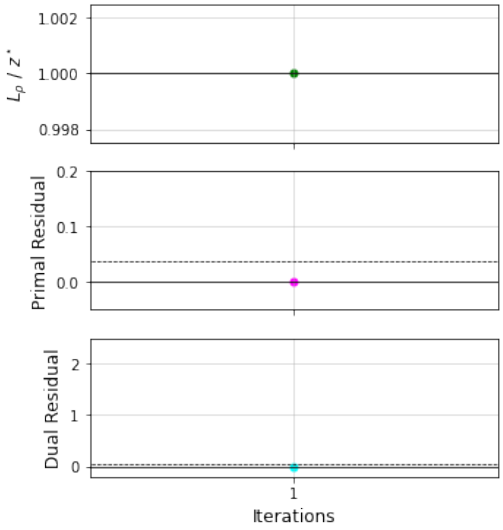
In Fig. 4.1, the convergence of objective value, and primal and dual tolerance for ADMM is presented. Each of the four selected instances is presented using three graphs. The top graph in green presents how the augmented Lagrangian value L_ρ changes for each iteration, normed with the optimal objective value $z^\star$ for the given instance. The solid black line at $L_\rho/z^\star = 1$ indicates where the augmented Lagrangian value $L_\rho = z^\star$. The middle graph in pink presents how the primal residual changes in each iteration and the dashed black line represents the primal

tolerance for the given instance. The bottom graph in blue present how the dual residual changes in each iteration and the dashed black line represents the dual tolerance for the given instance.

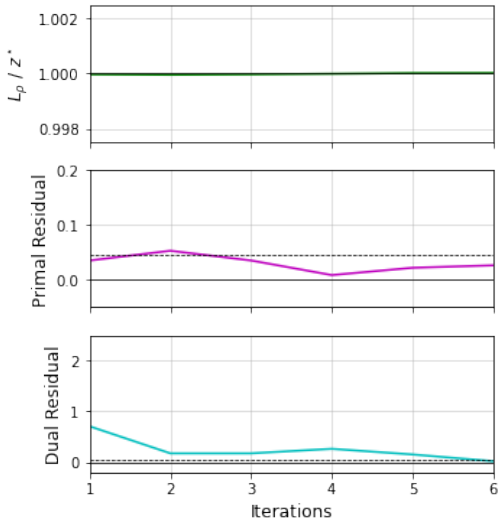
For instance 14_18_100, see Fig. 4.1a, ADMM terminated at 10 iterations. When termination occurred, ADMM had reached its optimal value $L_\rho = z^\star$. For instance 20_18_200, see Fig. 4.1b, ADMM terminated at 1 iteration. When termination occurred, ADMM had reached its optimal value $L_\rho = z^\star$. For instance 23_24_200, see Fig. 4.1c, ADMM terminated at 6 iterations. When termination occurred, ADMM had reached its optimal value $L_\rho = z^\star$. For instance 34_38_100, see Fig. 4.1d, ADMM terminated at 10 iterations. When termination occurred, ADMM had not reached its optimal value $L_\rho \neq z^\star$.



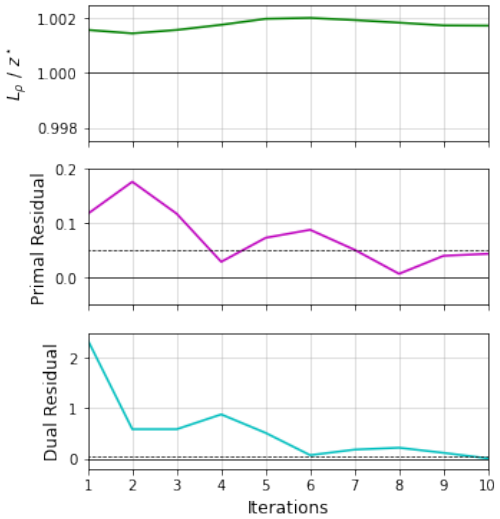
(a) 14_18_100



(b) 20_18_200



(c) 23_24_200



(d) 34_38_100

Figure 4.1: Normed objective value for ADMM, together with primal and dual residual for each iteration until termination. The dashed lines indicate primal and dual tolerances. Observe that the axis scales differ between the three plots.

In Fig. 4.2, ADMM instance 34_38_100 but with fixed $\rho = 1$. ADMM terminated after 13 iterations, when the optimal value $L_\rho = z^\star$ was reached.

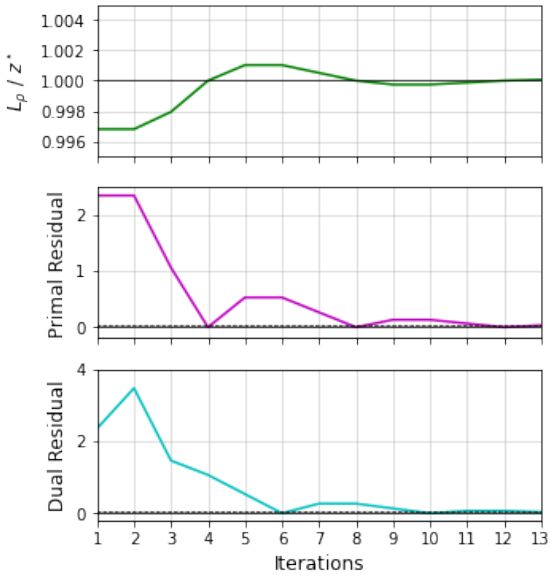


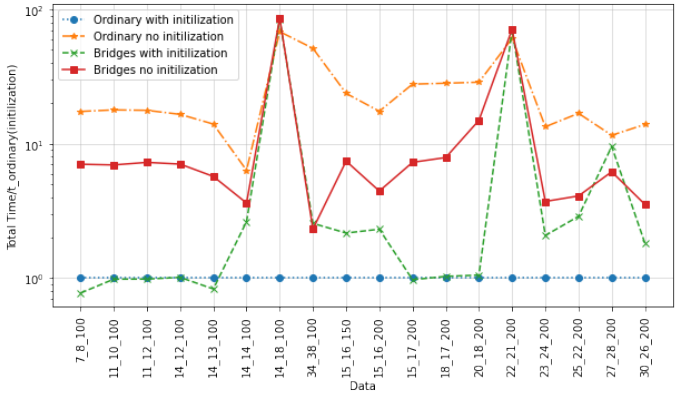
Figure 4.2: Convergens of ADMM for instance 34_38_100, with fixed $\rho = 1$.

Table 4.1 includes the mean and standard deviation of computing times in seconds for ADMM run on all 18 instances. It also includes the objective value and the number of iterations until the termination criteria was reached. The results are from 100 runs of the program on the same instance and time is the only thing that deviated between runs.

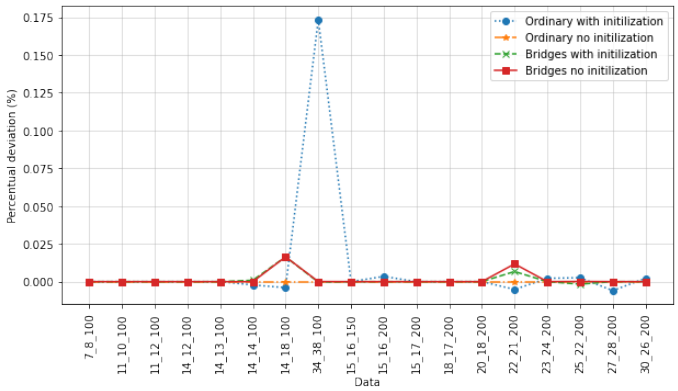
Data ($n_\mathcal{X}$_$n_\mathcal{Y}$_T)	Mean time (s)	Std time (s)	Objective Value	Iterations
7_8_100	0.0327	0.00416	377.416	1
11_10_100	0.0476	0.00428	716.306	1
11_12_100	0.0519	0.00788	607.404	1
14_12_100	0.0656	0.00860	885.590	1
14_13_100	0.0581	0.00555	664.185	1
14_14_100	0.150	0.00553	858.781	10
14_18_100	0.169	0.00910	1170.75	10
34_38_100	0.495	0.0120	2191.46	10
15_16_150	0.104	0.00729	988.633	1
15_16_200	0.246	0.00835	1083.50	6
15_17_200	0.138	0.00395	1249.32	1
18_17_200	0.152	0.00634	1082.22	1
20_18_200	0.180	0.0135	1266.17	1
22_21_200	0.480	0.0295	1637.46	10
23_24_200	0.396	0.00987	1708.82	6
25_22_200	0.418	0.0330	1445.26	6
27_28_200	0.730	0.0546	2055.19	10
30_26_200	0.550	0.0211	1723.17	6

Table 4.1: The mean and standard deviation of the runtime of 100 runs of ADMM on the eighteen instances.

In Fig. 4.3 we see the times and final values for four version of the ADMM, ADMM with and without initialization and bridges. In Fig. 4.3a we see the logarithmized final times normed by the time for ADMM with initialization and without bridges. In Fig. 4.3b the final values for the four ADMMs are presented. The values are presented as the percentual deviation from the optimal value. All algorithms except ADMM ordinary with initialization are run with fixed $\rho = 0.05$, while ADMM ordinary with initialization is run with normal varying ρ starting at $\rho = 20$. ADMM with bridges requires the number of bridges to be specified, we used $\sqrt{T/2}$ as the number of bridges for all instances. The blue dotted line with circles is the ordinary ADMM with initialization and no bridges, it is used as the standard for the time since it is the fastest over all, but it also deviates the most from the true objective, 0.175%. On the other hand, the orange dash-dotted line with stars is ordinary ADMM without initialization which is the slowest over all, but it is stable and finds the optimum. The green dashed line with "×" is ADMM with initialization and bridges which is as fast as ADMM ordinary with initialization for some instances and as slow as ADMM ordinary without initialization one some others. Finally, the last red solid line with squares is ADMM bridges without initialization which is slower than ADMM bridges with initialization and faster than ADMM without bridges and initialization.



(a) Relative values of computation times (time normalized by computation time for the model without bridges and with initialization). Logarithmic scale.



(b) Final relative objective values; percentual deviation from the optimal values.

Figure 4.3: The computation times and best objective values found for ADMM with and without bridges and initialization.

4.2 Results from the subgradient algorithm

In Fig. 4.4, the lower bound, upper bound, and Lagrangian objective values are presented for the subgradient method for each of the four selected instances until termination. The solid black lines indicate the optimal value $z^\star$. The red dash-dotted line represents the upper bound given by the heuristics. The thick blue line represents the best lower bound found, i.e. the highest value of the Lagrangian objective value. The thin green line at the bottom represents the Lagrangian objective value in each iteration.

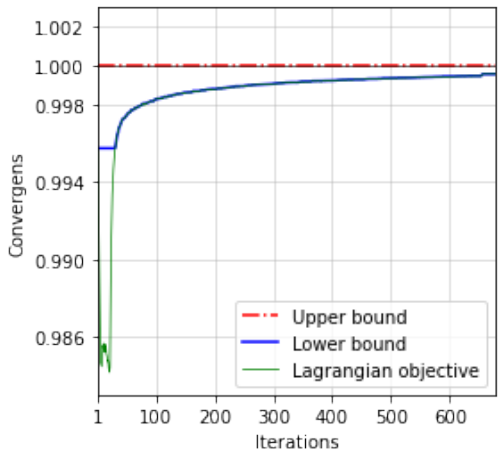
For instance 14_18_100, see Fig. 4.4a, the heuristic finds the optimal value; hence $\bar{z} = z^\star$. The lower bound is constant during the first 40 iterations. After 40 iterations, the Lagrangian objective value begins to improve its initial value, and starts to update the lower bound. After 724 iterations the subgradient method is terminated.

At termination the lower bound is 0.04% from the optimal value $z^\star$.

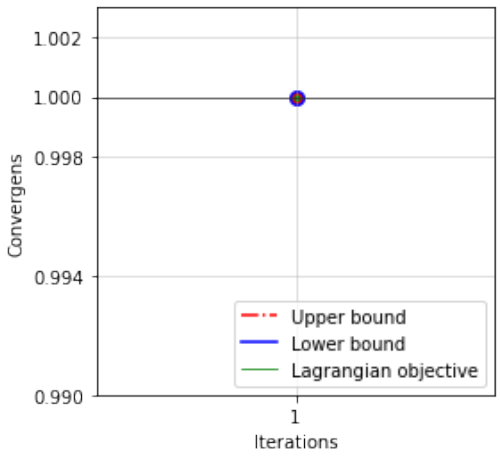
For instance 20_18_200, see Fig. 4.4b, the heuristic finds the optimal value; hence $\bar{z} = z^\star$. The Lagrangian objective value is the same as the upper bound in the first subgradient iteration and the subgradient method is therefore terminated. The optimal value $z^\star$ is found in the first iteration, since $\bar{z} = \underline{z}$.

For instance 23_24_200, see Fig. 4.4c, the heuristic finds the optimal value, hence $\bar{z} = z^\star$. The lower bound is constant during all iterations, because the Lagrangian objective never finds a better value than the initial value. After 202 iterations the subgradient method is terminated. At termination the lower bound is 0.12% from the optimal value $z^\star$.

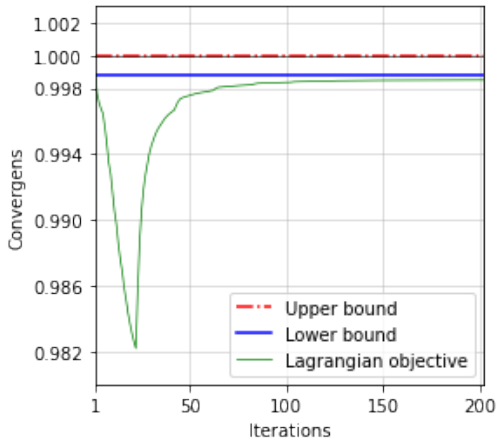
For instance 34_38_100, see Fig. 4.4d, the heuristic does not find the optimal value, hence $\bar{z} \neq z^\star$. The upper bound is 0.18% from the optimal objective value $z^\star$. The lower bound is constant during the first 40 iterations. After 40 iterations, the Lagrangian objective value begins to find better values and start to update the lower bound. After 1000 iterations the subgradient method is terminated. At termination the lower bound is 0.02% from optimal value $z^\star$.



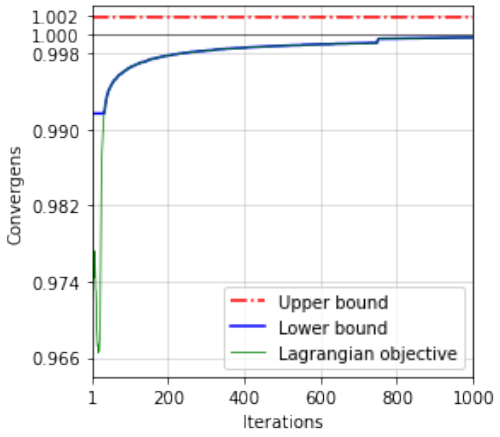
(a) Instance 14_18_100



(b) Instance 20_18_200



(c) Instance 23_24_200



(d) Instance 34_38_100

Figure 4.4: Normed convergence of upper and lower bound for four different instances, using the subgradient method. Note that the axis scales differ between instances.

In Table 4.2 the value for the heuristic — which is an upper bound on the objective value — and the subgradient value — which is a lower bound — are presented. It also includes the number of iterations it takes the respective method to generate the value, as well as the run time as a mean and standard deviation. The time for both the heuristic and the subgradient are parallelized with T threads, same with the total time presented in the table. The only times not parallelized are the unparallelized times which are the times for running the whole algorithm without parallelization.

The parallelized times are generated without implementing any parallelizing and are instead generated by running the code normally and only counting the longest time it takes to solve any of the assignment problems in that iteration. This will then approximate running the program with T number of threads. Both the heuristic and the subgradient method are parallelized, specifically, the T assignment problems are

parallelized in each iteration for both of them. The times are taken as the mean and standard deviation over 100 runs.

Data	Heuristic				Subgradient				Total Time (s)		Un parallelized Time (s)	
			Time (s)				Time (s)					
($n_{\mathcal{X}}$_$n_{\mathcal{Y}}$_T)	Value	Iterations	Mean	Std	Value	Iterations	Mean	Std	Mean	Std	Mean	Std
7_8_100	377.416	91	0.0236	0.00600	377.416	1	0.0100	0.00125	0.0337	0.00690	0.573	0.0460
11_10_100	716.306	96	0.0393	0.00350	716.306	1	0.0226	0.00185	0.0619	0.00473	0.686	0.0216
11_12_100	607.404	94	0.0386	0.00494	607.404	1	0.0212	0.00163	0.0599	0.00569	0.702	0.0357
14_12_100	885.590	95	0.0491	0.00420	885.590	1	0.0331	0.00238	0.0822	0.00609	0.757	0.0337
14_13_100	664.185	93	0.0478	0.00337	664.185	1	0.0298	0.00166	0.0776	0.00424	0.794	0.0241
14_14_100	858.800	87	0.0503	0.00775	858.302	142	0.147	0.0189	0.197	0.0248	1.91	0.118
14_18_100	1170.80	88	0.0605	0.00972	1170.30	724	0.704	0.0711	0.765	0.0759	6.95	0.345
34_38_100	2191.67[a]	100	0.267	0.0275	2187.26	1000	4.16	0.392	4.44	0.406	22.3	1.28
15_16_150	988.633	141	0.0964	0.0142	988.137	119	0.186	0.0177	0.282	0.0283	3.32	0.188
15_16_200	1083.46	197	0.127	0.0202	1082.93	390	0.673	0.0760	0.800	0.0926	9.90	0.804
15_17_200	1249.32	187	0.135	0.0246	1249.32	1	0.0728	0.00663	0.208	0.0298	3.23	0.241
18_17_200	1082.22	189	0.151	0.0178	1082.22	1	0.0835	0.00890	0.235	0.0235	3.52	0.205
20_18_200	1266.17	200	0.167	0.0100	1266.17	1	0.0964	0.00537	0.263	0.0138	3.85	0.109
22_21_200	1637.55	190	0.207	0.0274	1635.28	1000	3.32	0.309	3.53	0.328	26.6	1.57
23_24_200	1708.79	181	0.230	0.0186	1706.79	202	0.783	0.0414	1.01	0.0531	9.56	0.335
25_22_200	1445.23	197	0.223	0.0337	1443.23	202	1.00	0.104	1.23	0.125	9.53	0.576
27_28_200	2055.31	200	0.361	0.0461	2051.40	835	6.35	0.692	6.71	0.725	33.0	2.02
30_26_200	1723.13	195	0.357	0.0210	1721.81	225	1.66	0.110	2.02	0.121	18.7	0.540

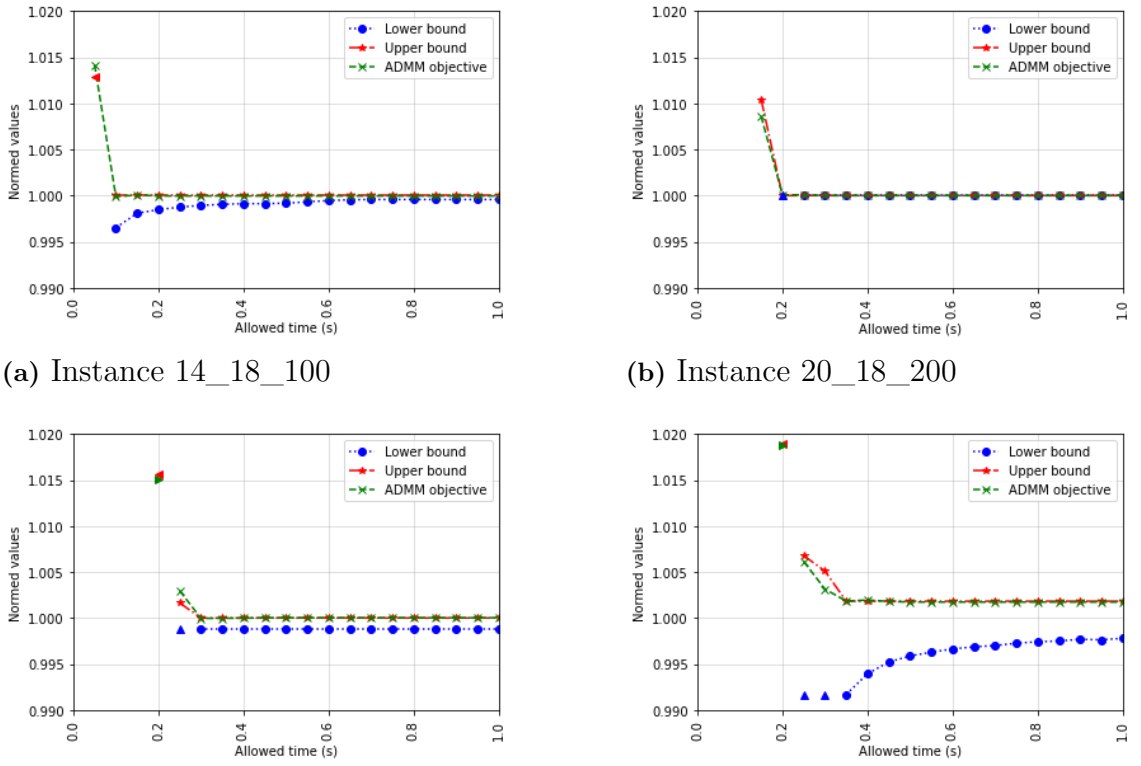
Table 4.2: The mean and standard deviation of the run time of 100 runs of the subgradient with parallelizing on different instances. Observe that the heuristic value is the true optimal value for all instances besides 34_38_100.

[a]The optimal objective value is 2187.67

4.3 A comparison between the ADMM and subgradient optimization

In Fig. 4.5 the mean values for 10 runs generated by the subgradient method and ADMM are presented when terminating after a certain computation time. The x-axis states how long time in seconds the methods are allowed to run before they are interrupted. If the time is too short, the algorithms will not generate a value. For some time limits, the algorithms sometimes generated values and sometimes not; in the figures these values (when they exist) are represented by triangles. The triangles represent the mean values of the values that were generated within that time. All values are normed by the actual objective value for the problem.

The red line with stars represents the upper bound for the subgradient in Fig. 4.5, in all four instances, it can be seen to decrease until it stabilizes at some value. The upper bound stabilizes at the optimal objective value for all instances except 34_38_100 where it finds a suboptimal upper bound. The green line with "×" represents the ADMM objective that utilizes the same heuristic that finds the upper bound. The heuristic is used as an initial guess in ADMM, the red (upper bound) and green (ADMM) lines can be seen to coincide a lot due to this. Finally, the



(a) Instance 14_18_100

(b) Instance 20_18_200

(c) Instance 23_24_200

(d) Instance 34_38_100

Figure 4.5: The values generated by subgradient and ADMM after some allowed time.

blue line with circles represents the lower bound from the subgradient method.

Note that the first values for the lower bound take longer time to generate than the upper bound and the ADMM objective. This is since the subgradient method that produces the lower bound requires an upper bound in order to choose a step length, meaning the algorithm cannot begin to update the lower bound before it has an upper bound. The lower bound behaves a bit different for the four different instances; for the instances 14_18_100 and 34_38_100, see Figs. 4.5a and 4.5d, the lower bound starts at some smaller value and increases towards the optimum value. For instance 20_18_200 (see Fig. 4.5b) on the other hand, the initial lower bound equals the optimal value, and optimality is verified as soon as the initial lower bound is computed. Finally for the instance 23_24_200 (see Fig. 4.5c), the first lower bound detected is suboptimal and is never updated in the allowed computing time.

The final values from the ADMM and the subgradient method can be found in Fig. 4.6, where the objective value for ADMM and the upper and lower bound for the subgradient are presented for all 18 instances. All values are normed by the optimal value for that instance. The green dashed line with "×" is the objective value of ADMM at termination; it finds the optimal value for all instances except instance 34_38_100. The red dash-dotted line with stars is the upper bound from the subgradient method; like the ADMM, it finds the optimal value for all instances except 34_38_100. The blue dotted line with circles is the lower bound from the subgradient method; compared to the others, which only miss the optimal value for one of the eighteen instances, it misses it a total of ten times.

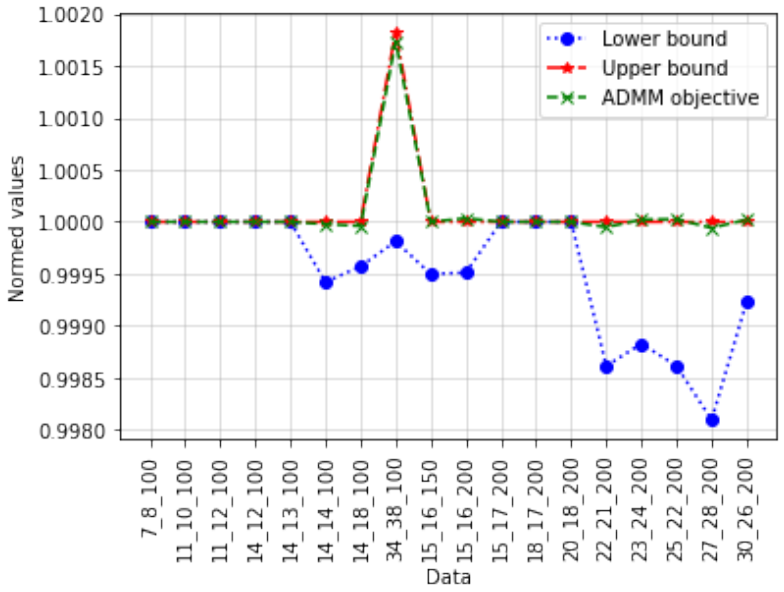


Figure 4.6: Normed values for obtained upper and lower bounds with subgradient and ADMM objective values, for all eighteen instances.

In Fig. 4.7 the total time for all 18 instances for unparallelized and parallelized subgradient method, as well as ADMM can be found. The total time is normed with the time for ADMM for each individual instance, and then and logarithmized. The

red solid line with dots at the top represents the time for unparallelized subgradients. The blue dash-dotted line with triangles in the middle represents the time for parallelized subgradients. Finally, the dashed green line with stars at the bottom represents the time for ADMM.

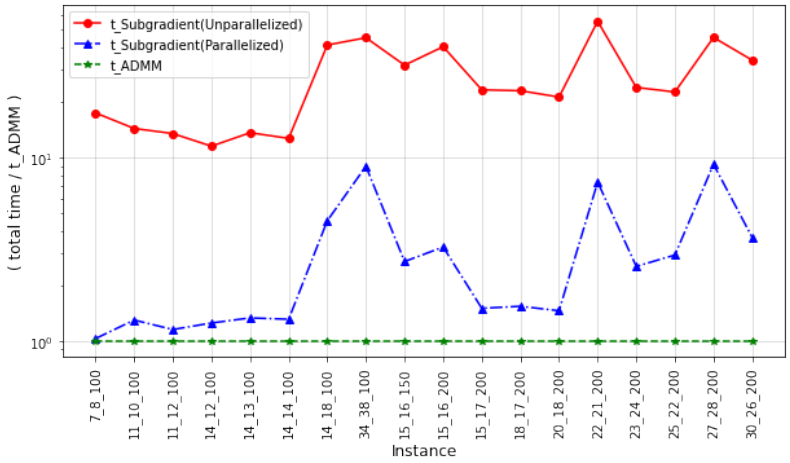
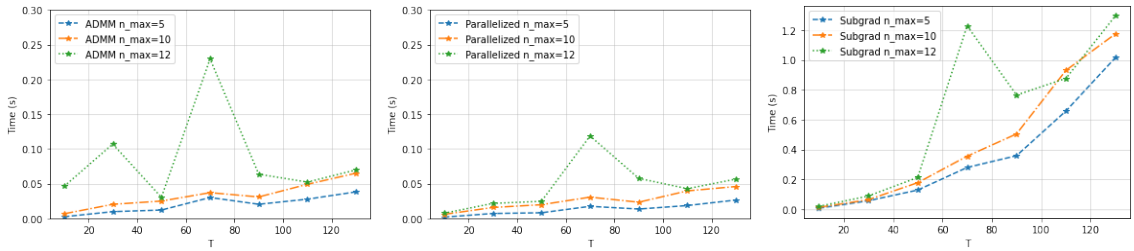


Figure 4.7: The total mean time for ADMM and unparallelized as well as parallelized subgradient method, normed with the total mean time of ADMM for each instance. All values are logarithmic.

In Fig. 4.8 the time performance is presented for the algorithms as a function of the number of time steps T for three values of $n_{\max}$, where $n_{\max}$ is the maximum of $n_{\mathbf{X}}$ and $n_{\mathbf{Y}}$. Note that the instance with $n_{\max} = 12$ and $T = 70$ takes a long time to solve for all three methods. Each value is the mean from ten runs of the algorithm at hand applied to the respective instance.



(a) ADMM **(b)** Parallelized subgradient **(c)** Unparallelized subgradient

Figure 4.8: Plots of how the time of each method changes with the value of T.

5

Discussion

5.1 Heuristic

For all of the eighteen + twenty-one instances, the heuristic found the optimal value within T iterations for all instances, besides 34_38_100. This indicates that the heuristic works well for the problem at hand. It is indeed a heuristic and not an optimizing algorithm, since we found an example where it did not find the optimal value. There were some instances where both algorithms found the optimal value in one iteration. Here the heuristic found the optimal value and then either ADMM or the subgradient method can be used to verify that the objective value from the heuristic is indeed the optimal value. For these cases, a large proportion of the total time is used to run the heuristic; see Table 4.2. Although, since we knew the optimal value from the heuristic, we were able to break after one iteration, meaning it is still faster to use the heuristic, see Fig. 4.3a.

For subgradient, we tried with a simpler heuristic which provided us with a worse upper bound, and use an ergodic sequence to update and improve the upper bound. The simple heuristic is essentially our heuristic described in Section 3.1, evaluated in the first iteration, which will be faster. For this combination, we needed more iterations before reaching the same upper bound as for the iterative heuristic. Hence, it was better to run the iterative heuristic instead of using an ergodic sequence. This was true even for the instance 34_38_100, where the iterative heuristic did not find the optimal value. For the ergodic sequence, it could find a better solution than the heuristic, but it had to run for more than 1000 iterations to find a better upper bound.

The heuristic seems promising in that it almost always finds the optimal value and it might be possible to build on it, in order to not be a heuristic. Another positive thing is that the heuristic can be parallelized over each time step k, resulting in a time complexity of $\mathcal{O}(Tn^3)$. The time complexity comes from the fact that in each iteration one must solve T assignment problems, where each assignment problem has a time complexity of $\mathcal{O}(n^3)$. The heuristic must run at most T iterations to converge, resulting in a total time complexity of $\mathcal{O}(T^2n^3)$ which can be parallelized in order to reduce it by a factor of T, to $\mathcal{O}(Tn^3)$.

5.2 ADMM

For ADMM, we can see that the best implementation according to speed is ordinary ADMM with initialization using the heuristic. In Fig. 4.3a, we can see that the fastest algorithms where the two algorithms with initialization, ordinary ADMM with initialization is the fastest for ten out of eighteen instances. For two instances, 7_8_100 and 14_13_100 ADMM bridges with initialization were the fastest, while the six remaining instances were solved equally fast for the two implementations with initialization. We can also see that if we compare the implementations without initialization, ADMM with bridges is the fastest. But, if we compare the implementations with initialization, bridges takes longer times to converge.

We could also see that for implementation with initialization, it was good from a speed perspective to start with a large value on ρ and vary it during the run. If we had bridges in the implementation, we needed to have a fixed small value for the implementation to converge.

When it comes to precision, ordinary ADMM with initialization is not always the best, as there is no guarantee that it converges. As we can see in Fig. 4.3b, the values from this implementation deviates from the optimal value more than for the other implementations.

We next focus on the four selected instances in Fig. 4.1, where we are using the heuristic as initialization. For instance 14_18_100 in Fig. 4.1a we can see that we start with a small primal residual, which indicates that the solution is primal feasible. The dual residual on the other hand starts a little bit higher. The dual residual tells us how good the objective value is. We can also see that when the primal residual is high, the dual residual is low. This pattern follows for each iteration, where they alternate between high and low values. For the objective value we can see that over the ten iterations, it is always close to the optimal value. For some iterations, it is a little bit over, and for some, it is a little bit under. ADMM finds the optimal value fast, but takes time to verify that it is optimal.

For instance 20_18_200 in Fig. 4.1b we can see that both the primal and the dual residual is zero after one iteration, which means that the optimal value is reached. We can also see that the objective value has reached the optimal value.

For instance 23_24_200 in Fig. 4.1c it is similar to Fig. 4.1a. In this figure, both primal and dual residual start at lower values. The alternating pattern is the same for this instance as for 14_18_100. The objective value is constantly at the optimal value, but ADMM takes time to verify it is optimal.

For instance 34_38_100 in Fig. 4.1d we can see that the objective value does not converge to the optimal value, although primal and dual residual are close to zero. For this instance, we saw that a too high value of the parameter ρ caused ADMM to converge to the wrong objective value. For smaller values of ρ the instance converges to the optimal value, but takes longer time to converge. This makes sense since the theory states that multi-block ADMM might not converge at all or may converge

for some small value of ρ. We can see that instance 34_38_100 converges when we reduce ρ, which means we have not encountered any instance where multi-block ADMM does not converge for our problem. However, when reducing ρ in order for ADMM to converge, it takes longer time until it terminates.

Another good thing about ADMM is the scalability. From Fig. 4.8a we can see that the time complexity for ADMM appears to be $\mathcal{O}(T)$ with respect to the number of time steps T. We also know from the theory about assignment problem solvers that their complexity is $\mathcal{O}(n^3)$. This means that the total complexity for the ADMM algorithm is likely to be $\mathcal{O}(Tn^3)$, where $n = n_{\mathcal{X}} + n_{\mathcal{Y}}$. Note in Fig. 4.8a that the instance with $n_{\max} = 12$ and $T = 70$ is slow for the subgradient as well as ADMM. The fact that the instance performs bad on multiple algorithms indicates that the long run time has something to do with the instance structure and not the algorithms. It is probable that both algorithms perform badly on some specific structure this instance possesses and it might be possible to make some changes such that the algorithms perform better on them. However, this could result in longer computation times for the other instances. There is also the instance with $n_{\max} = 12$ and $T = 30$ that ADMM performs bad on, but not the subgradient method. This could indicate that there is some structure in this instance that ADMM performs badly on, but not the subgradient method, indicating that the two algorithms might perform well on differently structured problem instances. There might be problem instances where one algorithm performs better and another instance with the same values of $n_{\mathcal{X}}$, $n_{\mathcal{Y}}$, and T where the other algorithm performs better. The algorithms might perform better on different structures, but there is still some structure where neither performs good. This will be discussed more in Section 5.4.

5.3 Subgradient method

For the subgradient method, we can see in Table 4.2 that for eight out of eighteen instances we can confirm the optimal value in one iteration, since the upper bound and lower bound are the same. One of these is 20_18_200, which we can see in Fig. 4.4b.

For two out of eighteen instances, we never find a better lower bound, and terminate at 202 iterations, since α is too small. One of these is 23_24_200, which we can see in Fig. 4.4c. Here we can see that the lower bound in the first iteration is the best. The Lagrangian objective value overshot in the beginning. At 20 iterations it begins to improve the Lagrangian objective value in every iteration, but too slow.

For two out of eighteen instances, we break at the maximum number of iterations (1000). One of these is 34_38_100, which we can see in Fig. 4.4d. After 40 iterations, the Lagrangian objective starts to update the lower bound. When it terminated at 1000 iterations, the lower bound is $1.9 \cdot 10^{-2}$ % from $z^\star$; see Table 4.2. Since the heuristic for this instance never found the optimal value, and the Lagrangian objective continues to improve the lower bound, the subgradient method continues to run.

The remaining six instances break at different numbers of iterations. One of these is 14_18_100, which we can see in Fig. 4.4a. This instance terminates after 724 iterations because the percentual increase of the lower bound is smaller than 10^{-6}. This means that we do update the lower bound, but the update is small, and the lower bound is presumably close to the optimal value. When termination occurs, the lower bound is $4.3 \cdot 10^{-2}$ % from $z^\star$; see Table 4.2. This shows that we are close to the optimal value when terminating.

For scalability, we can see in Fig. 4.8c that the subgradient seem to scale with $\mathcal{O}(T^2)$. But, if we can parallelize over T threads, then we can reduce the subgradient to scale with $\mathcal{O}(T)$, as can be seen in Fig. 4.8b. We can then use the fact that the assignment problem solver has $\mathcal{O}(n^3)$ and get that the total time complexity for the parallelized subgradient is likely $\mathcal{O}(Tn^3)$.

5.4 ADMM versus subgradient

From what we can see, neither algorithm is uniformly better than the other, but they seem to perform differently on some instances. Which algorithm to choose comes down to what the user wishes to achieve; if solution speed is important then ADMM might be better. One can even use only the heuristic which is the fastest, but with no quality guarantee on the solution. There is also a lot one can change regarding speed and accuracy by changing the hyperparameters. The hyperparameters used in the termination criteria for the two algorithms are good indicators of whether speed or precision is favored, meaning the tolerances. If the termination criteria are slack then the algorithm favors speed over accuracy. On the other hand, if the termination criteria are strict then accuracy is favored over speed. From Fig. 4.8 we can conclude that the two algorithms have the same time complexity $\mathcal{O}(Tn^3)$, meaning that the algorithms will be equally affected by the size of the problem instances. These results hold if one is able to parallelize the two algorithms, if that is not the case then the total time complexity for both algorithms will be $\mathcal{O}(T^2n^3)$ because of the heuristic. More specifically, without parallelizing the heuristic will have time complexity $\mathcal{O}(T^2n^3)$, while the ADMM part will have time complexity $\mathcal{O}(Tn^3)$ and the subgradient part $\mathcal{O}(T^2n^3)$. Meaning that for ADMM it is the preprocessing that has a higher-order time complexity, indicating that ADMM will still be faster than the subgradient when they are not parallelized. This comes from the fact that the two algorithms have the same preprocessing, but the main algorithm has different time complexity. In this case, ADMM is the better choice for speed.

There is also a difference in the output from the two algorithms; the subgradient returns two values while ADMM returns one. Both outputs can be used to quantify the performance of different MOT algorithms. This is provided that the gap between the lower and upper bound is small enough, if the gap is large then multiple MOT algorithms might overlap and it may be hard to say which is the best. On the other hand, there is no guarantee that ADMM converges to the correct value. We have seen that for small values of ρ, then all instances of our problem converge to the

optimal value. It might be that there exists some range for ζ where every instance of our problem converges for some small ρ. Even if that is not the case, we see that the only instance that did not converge for a large ρ is 34_38_100. Every other instance generated converged for large ρ as well as small, meaning there might be some correlation between the difficulty of the instance and the convergence.

We can also note in Fig. 4.6 that ADMM finds the correct value for almost every instance while the lower bound from subgradient is a bit away from optimal value for every instance except the ones that converge in one iteration. The subgradient is quick to converge for simpler instances, but for harder instances it takes time to converge. Furthermore, from Fig. 4.4 we can see that the rate of convergence for the subgradient decreases heavily with time. The accuracy of the two algorithms seems to depend on the underlying structure of the instance; it might be beneficial to study which structures the algorithms have a hard time solving. There are some instances like 11_12_100 and 20_18_200, for which both algorithms perform good, while both seem to have a hard time with instance 34_38_100.

To understand what seems to be a hard structure for the algorithms to solve, we selected two instances and some selected time steps to present how the ground truths and estimates move. This is presented more closely in Appendix B. In Fig. B.1 we can see the trajectories for some time steps of instance 34_38_100 and even when only considering one time step it is not clear which assignments are the best. There are some clusters in time steps 7 to 11 where essentially any ground truth could be assigned to any estimate. There are also some steps where switches are favorable, which might also increase the difficulty of the instance. This is because we can see that one estimate and one ground truth are close to each other for several time steps, but suddenly they diverge from each other. When diverging, a new estimate is born which is close to the old ground truth. This is a place where one probably wants to introduce switches, given that the selected switching cost is not too high. If one instead takes a look at the second instance in Fig. B.2 we can see the trajectories for some time steps of instance 23_24_200, for which ADMM converges to optimality while the subgradient method has a gap of 2: the upper bound is 1708.79 and the lower bound is 1706.79. This is the instance for which the lower bound is never updated, as seen in Fig. 4.4c. The lower bound is two units away from the optimal value which might indicate that there are two switches in the solution that are not suppose to be there. This means that there might be something in the structure which makes it hard for subgradient to find the optimal value. The structure is similar to instance 34_38_100, in that we have some clusters in time steps 22 to 26 but it involves fewer trajectories. However, two estimates are without a ground truth in the cluster. It could be that this level of cluster is hard for the subgradient but not for ADMM, while the cluster in Fig. B.1 is hard for both of them. It could also be that there is some other structure that makes the problems hard and it simply shows differently for ADMM and the subgradient. Many of the harder problems are the instances with a large number of $n_{\mathcal{X}}$ and $n_{\mathcal{Y}}$, however, we believe that the structures contribute more to the difficulty of the problem than the number of estimates and ground truths. A problem with large quantities of estimates and ground truths has a higher probability of having these hard structures, like clusters.

In order to identify which structures are hard, more structures have to be observed in order to make some concrete conclusion.

5.5 Integrality property

For all instances created, we have been able to verify that the optimal solution is integer. We tried for several weeks with a brute force to find a non-integer optimal solution, however without success. This made us think that the problem has the integrality property, although it is not totally unimodular. This made us come up with the conjecture about integrality property; see Section 2.3. Since we could not prove nor disprove it, it is still a conjecture.

If we would not have integrality property, then the connected 2D assignment problems would not have the integrality property. Instead, it would be optimal to assign fractional.

ADMM assumes that the problem possesses the integrality property, while the subgradient does not. If we cannot make this assumption then we will have to examine what solutions ADMM will give. In the implementation of ADMM, we use the knowledge that since the assignment problems are totally unimodular, it will take an integer value, either 0 or 1. Then, we can reduce the quadratic assignment problem in ADMM to a linear assignment problem. If the integrality property to the problem is lost, then this simplification will force the solution to take integer values, and not converge to the optimal value for the relaxed problem. The solution to ADMM will be integer and for this reason, it might be optimal in the original problem. There exists a theory about non-convex ADMM which one might be able to apply in this case and in that case the solution will be optimal [16]. If this holds then ADMM could be used even if the problem does not possess integrality property. If one cannot apply the theory about non-convex ADMM and wants to get the optimal value to the LP relaxation, then one will be forced to solve quadratic assignment problems instead of linear ones. This is much harder and will likely result in slower computations, meaning ADMM loses both speed and accuracy. The heuristic and subgradient method on the other hand make no assumptions about integrality property and will still yield an upper and a lower bound respectively.

Since we have only been able to make a conjecture about the integrality property, it would be beneficial to actually prove or disprove if the problem has the integrality property. In order to attempt to prove the property, we would suggest using the theory behind the proof of total unimodularity, as well as induction. There are many similarities between the theory about total unimodularity and our problem. As described in Section 2.3 we hope that by moving the condition from the constraint matrix to the vector, it is possible to prove the integrality property. This can be used to prove the property for a specific T, $n_{\mathcal{X}}$ and $n_{\mathcal{Y}}$, but we wish to prove it for any parameters and in order to do this we suspect that induction can be used. In order to disprove it, one simply has to find an instance where the optimal solution does not have integer variables and is not a linear combination of some integer solutions that are also optimal. Even if there exists some instance of the problem that does

not possess the integrality property, there are many that do and perhaps one can still prove that most instances possess the integrality property.

6

Conclusion

We can conclude that both multi-block ADMM and subgradient are able to quantify the performance of MOT algorithms. The two algorithms have different pros and cons when it comes to speed and accuracy. ADMM is generally the faster algorithm, while it cannot guarantee convergence to the optimal value. Although for sufficiently small values of ρ, the eighteen + twenty-one instances converge to optimal values at the expense of speed. For subgradient, the algorithm provides an upper and a lower bound where the optimal value is somewhere in the range between these two. If the difference between the upper and lower bound is large, it might be hard to differentiate MOT algorithms between each other. Provided that it is possible to parallelize the algorithms, they appeare to have equal computational complexity, namely $\mathcal{O}(Tn^3)$. In order to conclude that this is the computational complexity, one needs to test for more data points. The dominating factor contributing to the complexity of both algorithms is the solution of the 2D assignment problems. We can therefore conclude that both algorithms would benefit from faster assignment solvers.

To continue this research, we suggest to prove or disprove the integrality property for the problem, since it will impact how one can solve the problem. Since the assignment problems are the ones which contribute to the time complexity the most, looking for alternatives to the JVC algorithm to solve these could speed up the algorithms. With regards to ADMM, the main negative is that we cannot guarantee convergence; this is however an ongoing research. Since the heuristic works well for this problem, it would be interesting to see if one can modify it into an algorithm which guarantees an optimal value.

Bibliography

[1] Abu Sajana Rahmathullah, Angel F. Garcia-Fernandez, and Lennart Svensson. "Generalized optimal sub-pattern assignment metric". In: *2017 20th International Conference on Information Fusion (Fusion)* (July 2017). URL: http://dx.doi.org/10.23919/ICIF.2017.8009645.

[2] J. Lundgren, M. Rönnqvist, and P. Värbrand. *Optimization*. Studentlitteratur, 2010. ISBN: 978-91-44-05308-0.

[3] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. *Integer Programming*. Springer, 2014.

[4] L.G. Khachiyan. "Polynomial algorithms in linear programming". In: *USSR Computational Mathematics and Mathematical Physics* 20.1 (1980), pp. 53–72. ISSN: 0041-5553. DOI: https://doi.org/10.1016/0041-5553(80)90061-0.

[5] Hao Hu and Renata Sotirov. "The linearization problem of a binary quadratic problem and its applications". In: *Annals of Operations Research* 307 (2021), pp. 229–249. DOI: 10.1007/s10479-021-04310-x.

[6] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends® in Machine Learning* 3.1 (2011), pp. 1–122. URL: http://dx.doi.org/10.1561/2200000016.

[7] Dimitri P. Bertsekas. *Nonlinear Programming*. Third. Athena Scientific, 2016.

[8] Caihua Chen, Bingsheng He, Yinyu Ye, and Xiaoming Yuan. "The direct extension of ADMM for multi-block convex minimization problems is not necessarily convergent". In: *Mathematical Programming* 155 (2016), pp. 57–79. DOI: 10.1007/s10107-014-0826-5.

[9] Edvin Åblad, Ann-Brith Strömberg, and Domenico Spensieri. "Exact makespan minimization of unrelated parallel machines". In: *Open Journal of Mathematical Optimization* 2 (May 2021), pp. 1–15. DOI: 10.5802/ojmo.4.

[10] David F. Crouse. "On implementing 2D rectangular assignment algorithms". In: *IEEE Transactions on Aerospace and Electronic Systems* 52.4 (2016), pp. 1679–1696. DOI: 10.1109/TAES.2016.140952.

[11] Hoang Tuy. *Convex Analysis and Global Optimization*. Vol. 110. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-31484-6.

[12] Lennart Råde and Bertil Westergren. *Mathematics Handbook for Science and Engineering*. 5:18. Studentlitteratur, 2018.

[13] Nicholas John Higham. "Analysis of the Cholesky Decomposition of a Semidefinite Matrix". In: *Reliable Numerical Computation*. 1990.

[14] Paul Glaister. “75.45 Telescoping Series”. In: *The Mathematical Gazette* 75.474 (1991), pp. 443–445. DOI: https://doi.org/10.2307/3618631.

[15] Torbjörn Larsson, Michael Patriksson, and Ann-Brith Strömberg. “Conditional subgradient optimization — Theory and applications”. In: *European Journal of Operational Research* 88 (1996), pp. 382–403. DOI: https://doi.org/10.1016/0377-2217(94)00200-2.

[16] Yu Wang, Wotao Yin, and Jinshan Zeng. “Global Convergence of ADMM in Nonconvex Nonsmooth Optimization”. In: *Journal of Scientific Computing* 78 (2019), pp. 29–63. DOI: 10.1007/s10915-018-0757-z.

A

Dual Feasibility

The dual feasibility conditions can be found in (3.14) and here we will show the calculations for the other variables not shown in Section 3.2. We start with Φ^{s+1} which we know minimizes $L_\rho(W^{s+1}, H^{s+1}, \Phi, \Psi^s, \Lambda^s, \Xi^s)$ and therefore

$$\begin{aligned}
0 &\in \partial\Big(0 \cdot \phi_{k,i,j}^{s+1}\Big) + \lambda_{k,i,j}^s + \rho\Big(w_{k,i,j}^{s+1} - w_{k+1,i,j}^{s+1} - h_{k,i,j}^{s+1} + \phi_{k,i,j}^{s+1}\Big) \\
&= \partial\Big(0 \cdot \phi_{k,i,j}^{s+1}\Big) + \lambda_{k,i,j}^s + \rho\Big(w_{k,i,j}^{s+1} - w_{k+1,i,j}^{s+1} - h_{k,i,j}^{s+1} + \phi_{k,i,j}^{s+1}\Big) \\
&= \partial\Big(0 \cdot \phi_{k,i,j}^{s+1}\Big) + \lambda_{k,i,j}^{s+1}, \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}.
\end{aligned}$$

We find the same thing as for Ψ^{s+1} and Ξ^{s+1}, namely that Φ^{s+1} and Λ^{s+1} always satisfy (3.14e). The calculations for H can be found in Section 3.2, therefore we continue with W and start with (3.14c) where $k = T$ and get

$$\begin{aligned}
0 \in &\partial\Big(d_{T,i,j} w_{T,i,j}^{s+1}\Big) - \lambda_{T-1,i,j}^s + \xi_{T-1,i,j}^s \\
= &\partial\Big(d_{T,i,j} w_{T,i,j}^{s+1}\Big) - \lambda_{T-1,i,j}^s - \rho\Big(w_{T-1,i,j}^{s+1} - w_{T,i,j}^{s+1} - h_{T-1,i,j}^s + \phi_{T-1,i,j}^s\Big) \\
&+ \xi_{T-1,i,j}^s + \rho\Big(w_{T,i,j}^{s+1} - w_{T-1,i,j}^{s+1} - h_{T-1,i,j}^s + \psi_{T-1,i,j}^s\Big) \\
= &\partial\Big(d_{T,i,j} w_{T,i,j}^{s+1}\Big) - \lambda_{T-1,i,j}^s - \rho\Big(w_{T-1,i,j}^{s+1} - w_{T,i,j}^{s+1} - h_{T-1,i,j}^{s+1} + \phi_{T-1,i,j}^{s+1}\Big) \\
&+ \xi_{T-1,i,j}^s + \rho\Big(w_{T,i,j}^{s+1} - w_{T-1,i,j}^{s+1} - h_{T-1,i,j}^{s+1} + \psi_{T-1,i,j}^{s+1}\Big) \\
&- \rho\Big(h_{T-1,i,j}^{s+1} - h_{T-1,i,j}^s - \phi_{T-1,i,j}^{s+1} + \phi_{T-1,i,j}^s\Big) \\
&+ \rho\Big(h_{T-1,i,j}^{s+1} - h_{T-1,i,j}^s - \psi_{T-1,i,j}^{s+1} + \psi_{T-1,i,j}^s\Big) \\
= &\partial\Big(d_{T,i,j} w_{T,i,j}^{s+1}\Big) - \lambda_{T-1,i,j}^{s+1} + \xi_{T-1,i,j}^{s+1} \\
&+ \rho\Big(\phi_{T-1,i,j}^{s+1} - \phi_{T-1,i,j}^s - \psi_{T-1,i,j}^{s+1} + \psi_{T-1,i,j}^s\Big), \quad (i,j) \in \hat{\mathcal{I}}.
\end{aligned}$$

Where we find that (3.14c) is satisfied if and only if $\rho\Big(-\Phi_{T-1}^{s+1} + \Phi_{T-1}^{s} + \Psi_{T-1}^{s+1} - \Psi_{T-1}^{s}\Big) \in \partial\Big(d_{T,i,j} w_{T,i,j}^{s+1}\Big) - \lambda_{T-1,i,j}^{s+1} + \xi_{T-1,i,j}^{s+1}$. Similarly for W_k^{s+1}, we know it minimizes (3.14b) and get

$$
\begin{aligned}
0 \in & \partial\Big(d_{k,i,j} w_{k,i,j}^{s+1}\Big) + \lambda_{k,i,j}^{s} - \xi_{k,i,j}^{s} - \lambda_{k-1,i,j}^{s} + \xi_{k-1,i,j}^{s} \\
= & \partial\Big(d_{k,i,j} w_{k,i,j}^{s+1}\Big) + \lambda_{k,i,j}^{s} + \rho\Big(w_{k,i,j}^{s+1} - w_{k+1,i,j}^{s} - h_{k,i,j}^{s} + \phi_{k,i,j}^{s}\Big) \\
& - \lambda_{k-1,i,j}^{s} - \rho\Big(w_{k+1,i,j}^{s} - w_{k,i,j}^{s+1} - h_{k,i,j}^{s} + \psi_{k,i,j}^{s}\Big) \\
& - \xi_{k,i,j}^{s} - \rho\Big(w_{k-1,i,j}^{s+1} - w_{k,i,j}^{s} - h_{k-1,i,j}^{s} + \phi_{k-1,i,j}^{s}\Big) \\
& + \xi_{k-1,i,j}^{s} + \rho\Big(w_{k,i,j}^{s} - w_{k-1,i,j}^{s+1} - h_{k-1,i,j}^{s} + \psi_{k-1,i,j}^{s}\Big) \\
= & \partial\Big(d_{k,i,j} w_{k,i,j}^{s+1}\Big) + \lambda_{k,i,j}^{s} + \rho\Big(w_{k,i,j}^{s+1} - w_{k+1,i,j}^{s+1} - h_{k,i,j}^{s+1} + \phi_{k,i,j}^{s+1}\Big) \\
& - \xi_{k,i,j}^{s} - \rho\Big(w_{k+1,i,j}^{s+1} - w_{k,i,j}^{s+1} - h_{k,i,j}^{s+1} + \psi_{k,i,j}^{s+1}\Big) \\
& - \lambda_{k-1,i,j}^{s+1} - \rho\Big(w_{k-1,i,j}^{s+1} - w_{k,i,j}^{s+1} - h_{k-1,i,j}^{s+1} + \phi_{k-1,i,j}^{s+1}\Big) \\
& + \xi_{k-1,i,j}^{s+1} + \rho\Big(w_{k,i,j}^{s+1} - w_{k-1,i,j}^{s+1} - h_{k-1,i,j}^{s+1} + \psi_{k-1,i,j}^{s+1}\Big) \\
& + \rho\Big(w_{k+1,i,j}^{s+1} - w_{k+1,i,j}^{s} + h_{k,i,j}^{s+1} - h_{k,i,j}^{s} - \phi_{k,i,j}^{s+1} + \phi_{k,i,j}^{s}\Big) \\
& - \rho\Big(- w_{k+1,i,j}^{s+1} + w_{k+1,i,j}^{s} + h_{k,i,j}^{s+1} - h_{k,i,j}^{s} - \psi_{k,i,j}^{s+1} + \psi_{k,i,j}^{s}\Big) \\
& - \rho\Big(w_{k,i,j}^{s+1} - w_{k,i,j}^{s} + h_{k-1,i,j}^{s+1} - h_{k-1,i,j}^{s} - \phi_{k-1,i,j}^{s+1} + \phi_{k-1,i,j}^{s}\Big) \\
& + \rho\Big(- w_{k,i,j}^{s+1} + w_{k,i,j}^{s} + h_{k-1,i,j}^{s+1} - h_{k-1,i,j}^{s} - \psi_{k-1,i,j}^{s+1} + \psi_{k-1,i,j}^{s}\Big) \\
= & \partial\Big(d_{k,i,j} w_{k,i,j}^{s+1}\Big) + \lambda_{k,i,j}^{s+1} - \xi_{k,i,j}^{s+1} - \lambda_{k-1,i,j}^{s+1} + \xi_{k-1,i,j}^{s+1} \\
& + \rho\Big(2w_{k+1,i,j}^{s+1} - 2w_{k+1,i,j}^{s} + \phi_{k-1,i,j}^{s+1} - \phi_{k-1,i,j}^{s} - \psi_{k-1,i,j}^{s+1} + \psi_{k-1,i,j}^{s} \\
& - \phi_{k,i,j}^{s+1} + \phi_{k,i,j}^{s} + \psi_{k,i,j}^{s+1} - \psi_{k,i,j}^{s}\Big), \quad k \in \hat{\mathcal{K}}, \quad (i,j) \in \hat{\mathcal{I}}.
\end{aligned}
$$

Lastly, we do the same calculations for (3.14a) and get

$$
\begin{aligned}
0 \in & \partial\Big(d_{1,i,j} w_{1,i,j}^{s+1}\Big) + \lambda_{1,i,j}^{s} - \xi_{1,i,j}^{s} \\
= & \partial\Big(d_{1,i,j} w_{1,i,j}^{s+1}\Big) + \lambda_{1,i,j}^{s} + \rho\Big(w_{1,i,j}^{s+1} - w_{2,i,j}^{s} - h_{1,i,j}^{s} + \phi_{1,i,j}^{s}\Big) \\
& - \xi_{1,i,j}^{s} - \rho\Big(w_{2,i,j}^{s} - w_{1,i,j}^{s+1} - h_{1,i,j}^{s} + \psi_{1,i,j}^{s}\Big) \\
= & \partial\Big(d_{1,i,j} w_{1,i,j}^{s+1}\Big) + \lambda_{1,i,j}^{s} + \rho\Big(w_{1,i,j}^{s+1} - w_{2,i,j}^{s+1} - h_{1,i,j}^{s+1} + \phi_{1,i,j}^{s+1}\Big) \\
& - \xi_{1,i,j}^{s} - \rho\Big(w_{2,i,j}^{s+1} - w_{1,i,j}^{s+1} - h_{1,i,j}^{s+1} + \psi_{1,i,j}^{s+1}\Big) \\
& + \rho\Big(w_{2,i,j}^{s+1} - w_{2,i,j}^{s} + h_{1,i,j}^{s+1} - h_{1,i,j}^{s} - \phi_{1,i,j}^{s+1} + \phi_{1,i,j}^{s}\Big) \\
& - \rho\Big(- w_{2,i,j}^{s+1} + w_{2,i,j}^{s} + h_{1,i,j}^{s+1} - h_{1,i,j}^{s} - \psi_{1,i,j}^{s+1} + \psi_{1,i,j}^{s}\Big) \\
= & \partial\Big(d_{1,i,j} w_{1,i,j}^{s+1}\Big) + \lambda_{1,i,j}^{s+1} - \xi_{1,i,j}^{s+1} \\
& + \rho\Big(2w_{2,i,j}^{s+1} - 2w_{2,i,j}^{s} - \phi_{1,i,j}^{s+1} + \phi_{1,i,j}^{s} + \psi_{1,i,j}^{s+1} - \psi_{1,i,j}^{s}\Big), \quad (i,j) \in \hat{\mathcal{I}}.
\end{aligned}
$$

B

Trajectory plots

In Fig. B.1, the positions in the current time step, together with two previous time steps are visualized for some selected time steps of instance 34_38_100. The red dots represent estimates and the black stars represent ground truths. Blue stars represent newborn ground truths and green dots represent newborn estimates. The red dashed lines represent estimates trajectories in the previous two time steps, while the solid blacks line represents ground truths trajectories in the previous two time steps. These time steps are selected in order to present what structure is hard for the algorithms to solve. In time step $k = 4$, we can see that it is clear which to be assigned to each other and which to leave unassigned. In time step $k = 4$ to $k = 7$ one can think that ground truth and estimate at $[8, -5], k = 4$ shall be assigned to each other. But in $k = 8$ we can see that they start to diverge from each other. Then in the next time step $k = 9$ a new ground truth is born close to the previously discussed estimate. The new ground truth and previous estimate continue to follow each other well, while the old ground truth continues to diverge.

Similarly, at $k = 4$ there exists a ground truth and estimate at about $[7, 20]$. These two are near each other. In $k = 5$, these two start to diverge from each other. In $k = 6$, a new estimate is born, with the same distance to the ground truth as the previous estimate. In $k = 7$, the first estimate has disappeared, and the second estimate is close to the ground truth. The second estimate starts to diverge in $k = 8$ and continues to diverge in $k = 9$. In $k = 10$ a third estimate is born close to the ground truth, these two continue to follow each other well in $k = 11$.

Another difficulty, can be see in $k = 9-11$ is that there exist multiple ground truths and estimates in a small space $x = [0, 10]$, $y = [-40, 0]$. In this area, there exist multiple possible assignments and it is hard to know which is optimal.

As one can see by these examples, it is not always easy to know whether one should leave a ground truth or estimate unsigned, or introduce a track switch. These seems to be hard for the algorithms to solve.

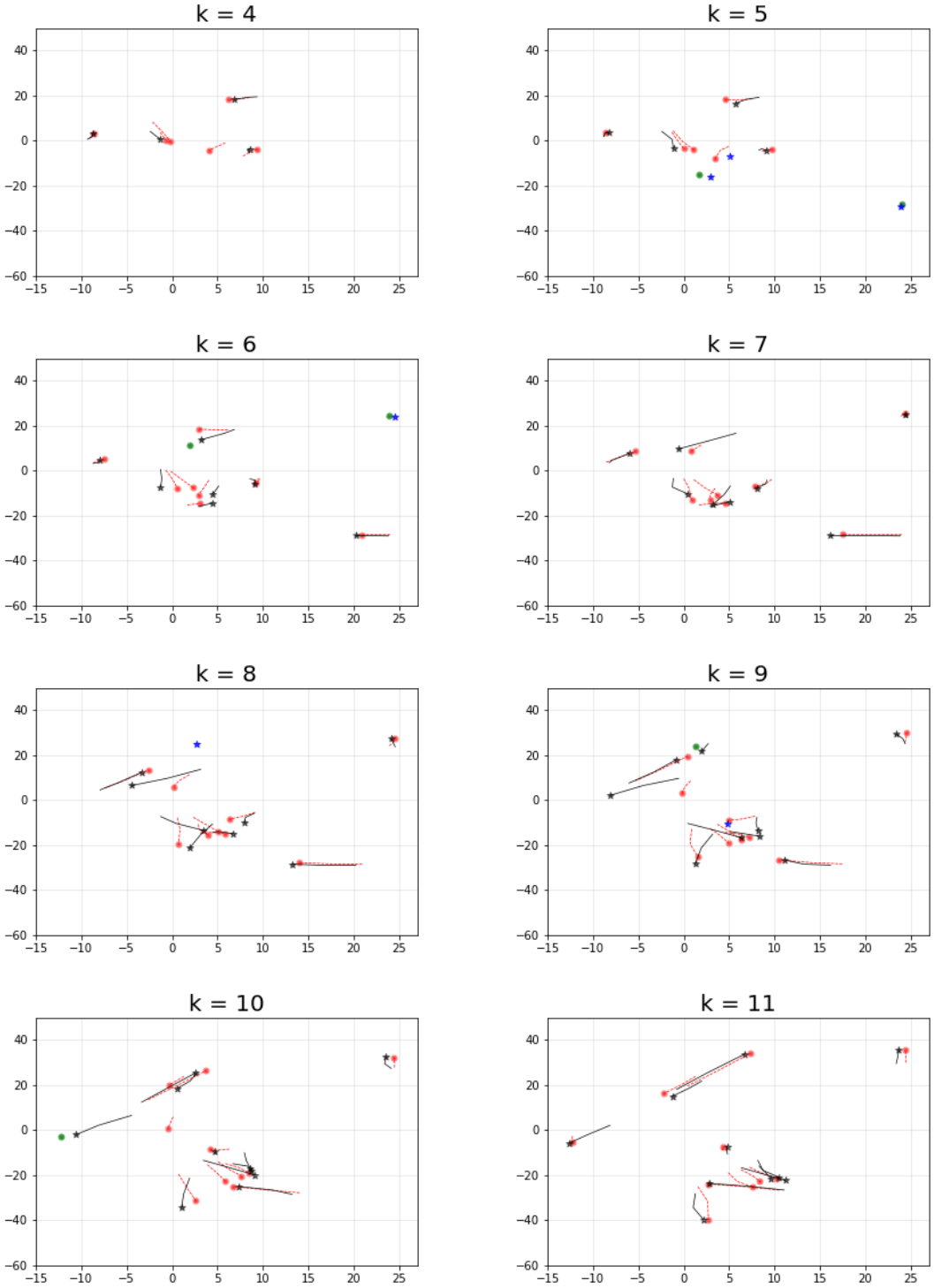


Figure B.1: Positions in the current time step, together with two previous time steps. Red dots represent estimates and black stars represent ground truths. Blue stars represent newborn ground truths and green dots represent newborn estimates.

In Fig. B.2, the positions in the current time step, together with two previous time steps are visualized for some selected time steps for instance 23_24_200. Red dots represent estimates and black stars represent ground truths. Blue stars represent newborn ground truths and green dots represent newborn estimates. Red dashed lines represent estimates trajectories, while solid black lines represent ground truths trajectories. For these figures, we want to show that a large number on $n_{\mathcal{X}}$ and $n_{\mathcal{Y}}$ does not necessarily indicate a hard problem.

A problem with the same number of ground truths $n_{\mathcal{X}}$ and estimates $n_{\mathcal{Y}}$ can have very different structures. The structure will depend on for example how many time steps a ground truth survives. If it only survives a few time steps, each individual assignment problem will be easier to solve, since fewer ground truths exist in every time step. Another example is how close the living ground truths and estimates are to each other. If multiple objects are close to each other, then it is harder to determine the optimal value, since there are multiple combinations. If the objects are far apart, there are fewer possible combinations which indicate an easier problem.

For this problem, one can see that there exist multiple ground truths and estimates within $x = [0, 20]$ and $y = [-20, 0]$, at $k = 21$. Over time, every ground truth follows an estimate well. For some single time steps, there exists an estimate which is closer to the ground truth than the estimate following well over time, if the switching penalty is small, then a track switch can be optimal for these cases. On the other hand, if the switching penalty is high, then the ground truth and estimate which follow each other well over time will be assigned. Other estimates will be assigned as false detected objects, although it is not necessarily optimal if you only evaluate a single time step as for GOSPA.

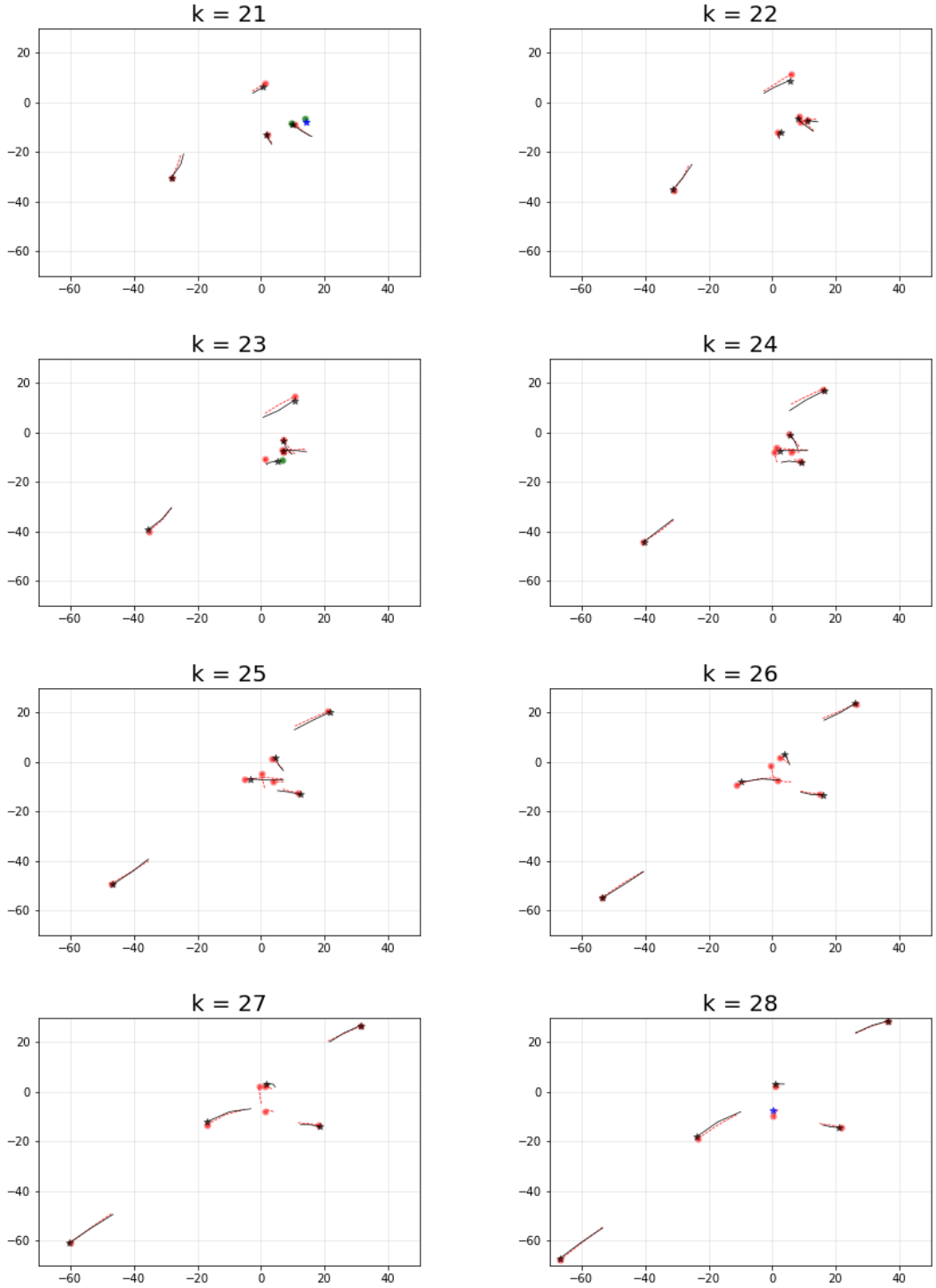


Figure B.2: Positions in the current time step, together with two previous time steps. Red dots represent estimates and black stars represent ground truths. Blue stars represent newborn ground truths and green dots represent newborn estimates.

C

Code

All functions used are appended below. The code implemented in Jupyter Notebook, together with all instances used can also be found on GitHub, `https://github.com/Camilla-C/sequentially-connected-2D-assignment-problems`.

[1]:
```
def Read_data(data):

    dataX = scipy.io.loadmat('X_' + data + '.mat')
    dataY = scipy.io.loadmat('Y_' + data + '.mat')
    X = dataX['temp']
    Y = dataY['temp']

    return X, Y
```

[2]:
```
def Create_D(X,Y,p,c):

    n_X = len(X[0,0])
    n_Y = len(Y[0,0])
    T = len(X[0])
    D = np.zeros((T,n_X+1,n_Y+1))
    cp = c ** p
    cp2 = cp/2

    for t in range(T):
        for i in range(n_X+1):
            X_exists = True
            if i == n_X:
                X_exists = False
            elif any(np.isnan(X[:,t,i])):
                X_exists = False

            for j in range(n_Y+1):
                Y_exists = True
                if j == n_Y:
                    Y_exists = False
                elif any(np.isnan(Y[:,t,j])):
                    Y_exists = False

                if X_exists and Y_exists:
                    D[t,i,j] = min(cp, (np.linalg.norm(X[:,t,i] -␣
↪Y[:,t,j], ord=p)) ** p)

                elif not X_exists and not Y_exists:
                    pass

                else:
                    D[t,i,j] = cp2

    return D
```

X

[3]:
```
def Update_W(D, beta):

    n_X = len(D) - 1
    n_Y = len(D[0]) - 1

    temp1 = D[0:n_X,0:n_Y] + beta
    temp2 = np.transpose(np.tile(D[0:n_X,n_Y], (n_X,1)))
    temp3 = np.tile(D[n_X,0:n_Y], (n_Y,1))
    temp4 = np.zeros((n_Y,n_X))

    cost = np.block([[temp1, temp2],[temp3, temp4]])

    row_ind, col_ind = linear_sum_assignment(cost)

    row_ind = [n_X if x>n_X else x for x in row_ind]
    col_ind = [n_Y if x>n_Y else x for x in col_ind]

    W = np.zeros((n_X + 1, n_Y + 1))
    W[row_ind,col_ind] = 1
    W[n_X,n_Y] = 0

    return W
```

[4]:
```
def Get_upper_bound(D, switching_penalty):

    ############ Get input and def constants␣
↪#####################################################
    n_X = len(D[0]) - 1
    n_Y = len(D[0,0]) - 1
    T = len(D)

    ############ Initilization␣
↪#####################################################################
    W = np.zeros((T, n_X+1, n_Y+1))

    for iterations in range(T):

        ############ update W␣
↪#####################################################################
        W_prev = np.copy(W)
        beta = switching_penalty * (np.ones((n_X, n_Y)) -␣
↪W_prev[1,0:n_X,0:n_Y] -10**-3)
        W[0,:,:] = Update_W(D[0,:,:], beta)

        for t in range(1,T-1):
```

```
            beta = switching_penalty * (np.ones((n_X, n_Y)) -␣
↪W_prev[t-1,0:n_X,0:n_Y] - W_prev[t+1,0:n_X,0:n_Y] -10**-3)
            W[t,:,:] = Update_W(D[t,:,:], beta)

        beta = switching_penalty * (np.ones((n_X, n_Y)) -␣
↪W_prev[T-2,0:n_X,0:n_Y] -10**-3)
        W[T-1,:,:] = Update_W(D[T-1,:,:], beta)

        ############ Calculate subgradients and objective value␣
↪######################
        if (W == W_prev).all():
            break

    switches = 0
    for t in range(0,T-1):
        switches += np.absolute(W[t,0:n_X,0:n_Y] - W[t+1,0:n_X,0:
↪n_Y]).sum()

    objective = (D * W).sum() + switching_penalty * switches

    return objective
```

[5]:
```
def Subgrad_Opt(X, Y, p, c, mu, max_iter):

    ############ Get input and def constants␣
↪#####################################################
    D = Create_D(X,Y,p,c)
    switching_penalty = (mu ** p)/2
    n_X = len(X[0,0])
    n_Y = len(Y[0,0])
    T = len(X[0])

    ############ Initilization␣
↪#########################################################################
    W = np.zeros((T, n_X+1, n_Y+1))
    dual_var = 0.5*switching_penalty*np.ones((T-1, n_X, n_Y))
    delta_W = np.zeros((T-1, n_X, n_Y))
    MDS = 0.5*np.ones((T-1, n_X, n_Y))

    ############### CALCULATE UPPER BOUND␣
↪#########################################
    upper_bound = Get_upper_bound(D, switching_penalty)
    lower_bound = -np.inf

    last_improvement = 0
```

```
    alpha = 0.7
    lagrangean_obj = lower_bound

    for iterations in range(max_iter):

        delta_lambda = 2*dual_var - switching_penalty
        ############ update W␣
↪#################################################################

        beta = delta_lambda[0,:,:]
        W[0,:,:] = Update_W(D[0,:,:], beta)

        for t in range(1,T-1):
            beta = -delta_lambda[t-1,:,:] + delta_lambda[t,:,:]
            W[t,:,:] = Update_W(D[t,:,:], beta)

        beta = - delta_lambda[T-2,:,:]
        W[T-1,:,:] = Update_W(D[T-1,:,:], beta)

        for t in range(1,T):
            delta_W[t-1,:,:] = W[t-1,0:n_X,0:n_Y] - W[t,0:n_X,0:
↪n_Y]

        ############ Calculate subgradients and objective value␣
↪######################
        subgradient = np.copy(delta_W)
        lagrangean_obj_prev = lagrangean_obj
        lagrangean_obj = (D * W).sum() +␣
↪(delta_lambda*subgradient).sum()

        ############  Check special cases for projection␣
↪###############################
        if (dual_var == 0).any() or (dual_var ==␣
↪switching_penalty).any():
            ind1 = np.argwhere(dual_var == 0)
            ind2 = np.argwhere(dual_var == switching_penalty)

            for i in ind1:
                if subgradient[i[0], i[1], i[2]] < 0:
                    subgradient[i[0], i[1], i[2]] = 0
                if MDS[i[0], i[1], i[2]] < 0:
                    MDS[i[0], i[1], i[2]] = 0

            for i in ind2:
                if subgradient[i[0], i[1], i[2]] > 0:
```

```
                    subgradient[i[0], i[1], i[2]] = 0
                if MDS[i[0], i[1], i[2]] > 0:
                    MDS[i[0], i[1], i[2]] = 0

        if (subgradient==0).all():
            lower_bound = lagrangean_obj
            break

        if (MDS==0).all():
            if lagrangean_obj > lower_bound:
                lower_bound = lagrangean_obj
            break

        ############ Modified Deflected Subgradient␣
↪####################################
        theta = max(0, -(subgradient*MDS).sum() / (np.linalg.
↪norm(subgradient)*np.linalg.norm(MDS)))
        sigma = 1/(2-theta)

        gamma_ADS = np.linalg.norm(subgradient)/np.linalg.norm(MDS)
        gamma_MGT = max(0, -sigma * (subgradient*MDS).sum() /␣
↪(MDS**2).sum())
        gamma_MDS = (1-theta) * gamma_MGT + theta * gamma_ADS
        MDS = subgradient + gamma_MDS * MDS

        step_size = alpha*(upper_bound - lagrangean_obj)/((MDS**2).
↪sum())

        dual_var = dual_var + step_size * MDS
        dual_var[dual_var<0] = 0
        dual_var[dual_var>switching_penalty] = switching_penalty

        ############### Terminate and update Alpha␣
↪#########################
        if lagrangean_obj > lower_bound:
            prev_lb = lower_bound
            lower_bound = lagrangean_obj
            if lower_bound - prev_lb < prev_lb*(10**-6):
                small += 1
            else:
                small = 0

            last_improvement = 0
```

```
        if ((upper_bound - lower_bound) < 0.5) or (alpha < 10**-3)␣
↪or (small == 3):
            break

        if last_improvement >= 20:
            alpha *= 0.5
            last_improvement = 0

        last_improvement += 1

    return lower_bound, upper_bound
```

[6]:
```
def Initilization(D, switching_penalty):

    ############ Get input and def constants␣
↪#####################################################
    n_X = len(D[0]) - 1
    n_Y = len(D[0,0]) - 1
    T = len(D)

    ############ Initilization␣
↪#####################################################################
    W = np.zeros((T, n_X+1, n_Y+1))
    H = np.zeros((T-1, n_X, n_Y))
    S_1 = np.zeros((T-1, n_X, n_Y))
    S_2 = np.zeros((T-1, n_X, n_Y))

    for iterations in range(T):

        ############ update W␣
↪#####################################################################
        W_prev = np.copy(W)

        beta = switching_penalty * (np.ones((n_X, n_Y)) -␣
↪W_prev[1,0:n_X,0:n_Y] - 10**-3)
        W[0,:,:] = Update_W(D[0,:,:], beta)

        for t in range(1,T-1):
            beta = switching_penalty * (np.ones((n_X, n_Y)) -␣
↪W_prev[t-1,0:n_X,0:n_Y] - W_prev[t+1,0:n_X,0:n_Y] - 10**-3)
            W[t,:,:] = Update_W(D[t,:,:], beta)

        beta = switching_penalty * (np.ones((n_X, n_Y)) -␣
↪W_prev[T-2,0:n_X,0:n_Y] - 10**-3)
        W[T-1,:,:] = Update_W(D[T-1,:,:], beta)
```

```
        ############ Calculate subgradients and objective value␣
↪#####################
        if (W == W_prev).all():
            break

    for t in range(T-1):
        H[t,:,:] = np.absolute( W[t,0:n_X,0:n_Y] - W[t+1,0:n_X,0:
↪n_Y] )
        S_1[t,:,:] = H[t,:,:] - W[t,0:n_X,0:n_Y] + W[t+1,0:n_X,0:
↪n_Y]
        S_2[t,:,:] = H[t,:,:] - W[t+1,0:n_X,0:n_Y] + W[t,0:n_X,0:
↪n_Y]

    return W, H, S_1, S_2
```

[7]:
```
def ADMM_ordinary(X, Y, p, c, mu, rho, max_iter):

    ### Get input and def constants ###
    D = Create_D(X,Y,p,c)
    switching_penalty = (mu ** p)/2
    n_X = len(X[0,0])
    n_Y = len(Y[0,0])
    T = len(X[0])

    tol = 10**-4
    primal_tol = tol*math.sqrt(n_X*n_Y*(T-1)*2)
    dual_tol = tol*math.sqrt(n_X*n_Y*(T-1)*3)

    ### Parameters for varing rho ###
    rho_incr = 2
    rho_decr = 2
    rho_tol = 10

    ### Initilization ###
    W, H, S_1, S_2 = Initilization(D, switching_penalty)

    lambda_1 = np.zeros((T-1, n_X, n_Y))
    lambda_2 = np.zeros((T-1, n_X, n_Y))

    r = np.zeros((T-1, 2, n_X, n_Y))

    for iterations in range(max_iter):

        ### update W ###
```

```
        W_prev = np.copy(W)
        delta_lambda = lambda_1[0,:,:] - lambda_2[0,:,:]
        delta_S = S_1[0,:,:] - S_2[0,:,:]
        rho_part = delta_S - 2*W[1,0:n_X,0:n_Y] + np.
↪ones((n_X,n_Y))
        beta = delta_lambda + rho*rho_part
        W[0,:,:] = Update_W(D[0,:,:], beta)

        for t in range(1,T-1):
            W_part = - W[t-1,0:n_X,0:n_Y] - W[t+1,0:n_X,0:n_Y] +␣
↪np.ones((n_X,n_Y))
            rho_part = - delta_S + 2*W_part
            beta = - delta_lambda + rho*rho_part
            delta_lambda = lambda_1[t,:,:] - lambda_2[t,:,:]
            delta_S = S_1[t,:,:] - S_2[t,:,:]
            beta = beta + delta_lambda + rho*delta_S
            W[t,:,:] = Update_W(D[t,:,:], beta)

        rho_part = - delta_S - 2*W[T-2,0:n_X,0:n_Y] + np.
↪ones((n_X,n_Y))
        beta = - delta_lambda + rho*rho_part
        W[T-1,:,:] = Update_W(D[T-1,:,:], beta)

        ### update H ###
        H_prev = np.copy(H)
        H = switching_penalty * np.ones((T-1,n_X,n_Y))
        H = H - lambda_1 - lambda_2 - rho * S_1 - rho * S_2
        H = - H/(2*rho)
        H[H<0] = 0

        ### update S_1, S_2, lambda_1 and lambda_2 ###
        S_1_prev = np.copy(S_1)
        S_2_prev = np.copy(S_2)
        for t in range(T-1):
            delta_W = W[t,0:n_X,0:n_Y] - W[t+1,0:n_X,0:n_Y]
            delta_W_H_1 = delta_W - H[t,:,:]
            delta_W_H_2 = - delta_W - H[t,:,:]

            ### update S_1 ###
            min_S_1 = - lambda_1[t,:,:]/rho - delta_W_H_1
            min_S_1[min_S_1<0] = 0
            S_1[t,:,:] = np.copy(min_S_1)

            ### update S_2 ###
            min_S_2 = - lambda_2[t,:,:]/rho - delta_W_H_2
```

```
            min_S_2[min_S_2<0] = 0
            S_2[t,:,:] = np.copy(min_S_2)

            ### update lambda_1 ###
            lambda_1[t,:,:] = lambda_1[t,:,:] + rho*(delta_W_H_1 +␣
↪S_1[t,:,:])

            ### update lambda_2 ###
            lambda_2[t,:,:] = lambda_2[t,:,:] + rho*(delta_W_H_2 +␣
↪S_2[t,:,:])

            r[t,0,:,:] = delta_W_H_1 + S_1[t,:,:]
            r[t,1,:,:] = delta_W_H_2 + S_2[t,:,:]

        primal_residual = np.linalg.norm(r)
        z = np.stack((2*W[1:T,0:n_X,0:n_Y], S_1, S_2))
        z_prev = np.stack((2*W_prev[1:T,0:n_X,0:n_Y], S_1_prev,␣
↪S_2_prev))
        dual_residual = rho*np.linalg.norm((z-z_prev))

        if primal_residual < primal_tol and dual_residual <␣
↪dual_tol:
            break

        if primal_residual > rho_tol*dual_residual:
            rho = rho_incr*rho
        elif dual_residual > rho_tol*primal_residual:
            rho = rho/rho_decr

    ### Calculate final value ###
    objective_value = (D * W).sum() + switching_penalty * H.sum()

    return objective_value
```

[ ]:
```
import numpy as np
import scipy.io

import math
from scipy.optimize import linear_sum_assignment

data = ['7_8_100', '11_10_100', '11_12_100']

p = 1
c = 20
mu = 2
```

```
rho = 20
max_iter = 1000
lower_bound = np.zeros((len(data)))
upper_bound = np.zeros((len(data)))
objective_value = np.zeros((len(data)))
di = 0

for dat in data:
    X, Y = Read_data(dat)
    print('Data: ', dat, '\n')

    lower_bound[di], upper_bound[di] = Subgrad_Opt(X, Y, p, c, mu,␣
 ↪max_iter)
    objective_value[di] = ADMM_ordinary(X, Y, p, c, mu, rho,␣
 ↪max_iter)

    di += 1
  ␣
 ↪print('-------------------------------------------------------')
```

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

CHALMERS
UNIVERSITY OF TECHNOLOGY