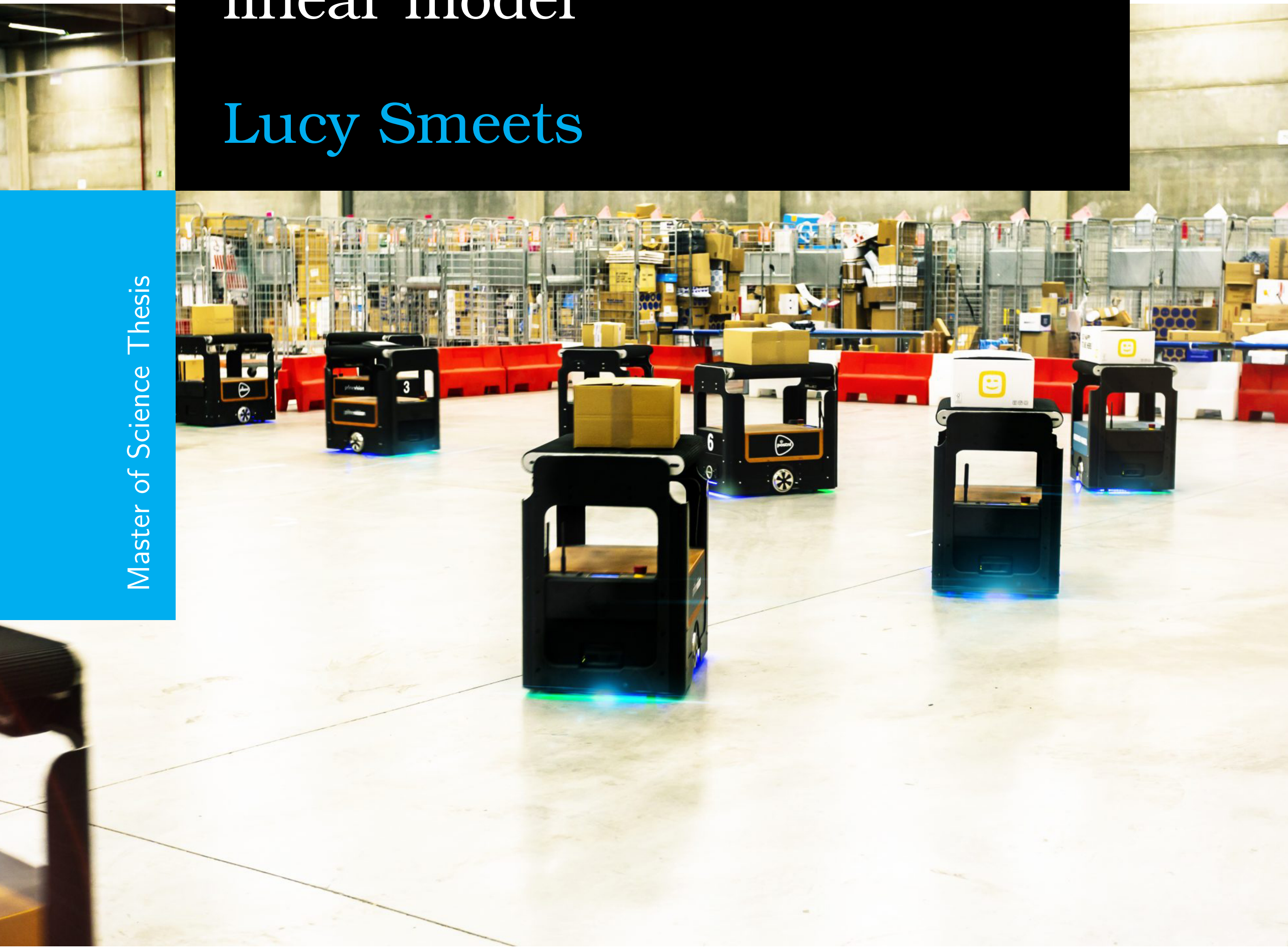
Modeling and scheduling an autonomous sorting system using a switching max-plus linear model

Lucy Smeets

Master of Science Thesis



TU Delft Delft University of Technology

primevision

Delft Center for Systems and Control

Modeling and scheduling an autonomous sorting system using a switching max-plus linear model

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Lucy Smeets

June 28, 2022

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

The work in this thesis was supported by Prime Vision BV. Their cooperation is hereby gratefully acknowledged.

Abstract

Sorting systems form an example of event driven systems. These types of systems are referred to as discrete event systems (DES), and they consist of jobs that need to be performed at available resources. In an autonomous sorting system, jobs consist of robots receiving and delivering parcels at the correct locations. With scheduling, optimal allocation of the jobs to those resources over time is computed, where the decisions that need to be made are routing, ordering and synchronization. The behaviour of DES is often described by non-linear models, but max-plus linear (MPL) systems are a class of DES that can be described by a model that is linear in the max-plus algebra. This algebra uses two operators maximization ($\oplus$) and addition ($\otimes$). Allowing different routes and switching between orders of jobs extends an MPL system to a switching max-plus linear (SMPL) system. Robots in a sorting system often have many routes to choose from, and need to make order choices with respect to other robots in the system.

In this thesis, a general SMPL model is made for the autonomous sorting system at software company Prime Vision, which can be applied to any sorting area design. The solution to the scheduling problem for the model results in a time schedule for the active robots at the correct locations in the sorting area, as well as the optimal decisions on routing, ordering and synchronization. The optimization problem is solved with a model predictive scheduling (MPS) approach and recast as a mixed integer linear programming (MILP) problem. The model is created in Python and the optimization problem is solved with Gurobi. The resulting schedule is visualized with a simulation, in which the decisions of the robots are clearly shown. An idea for implementation of the optimization into the sorting system is given as well.

Table of Contents

Acknowledgements

This document is the final deliverable for my Master of Science thesis. Before I started working on my thesis, I knew I wanted to conduct my thesis research at a company. During my years in Delft, I did not gain much working experience and I felt like graduating at a company would be a nice opportunity to do so. Right at the time I needed to define my project, my friend Umit asked me if I wanted to do an internship at the company he worked at, Prime Vision. He felt like my background in mathematics and systems and control could be of use in one of their innovations regarding navigation of automated guided vehicles. I turned to my thesis supervisor dr.ir. Ton van den Boom and asked for the possibilities of conducting my thesis research at Prime Vision. Ton was immediately excited, and after meeting with my company supervisor Mart Ruijs, we came up with an idea to use max-plus algebra for a sorting application, which would eventually become the subject of this thesis.

Many people around me warned me for the ups and downs that are a part of the thesis life. I was naive and thought I could plan better and work more well-structured than they did, especially after hearing their tips and tricks. Nonetheless, I experienced the same emotional roller coaster as anyone else. I started my thesis during lockdown with a curfew, in which I found it very hard to find focus. After the lockdown, it became even harder to concentrate on finishing a research that would mean the end of my student life. But here we are. As I am writing this, it becomes more and more clear to me that my thesis is actually coming to an end. And to my own surprise, I find myself to be excited to start my career, which I had been dreading for a long time during my thesis.

It might sound a bit dull, but I learned so much more during my thesis than I ever expected. The theory on max-plus algebra, which was completely new to me, and the experience of working in a different environment were the two things I expected to learn from. In addition, I learned some practical things like programming in Python, object-oriented programming and how to use Git. But also, I learned that I can do more than just study for exams and reproduce proofs and find solutions to difficult equations. Whenever I got stuck or I needed to do something which was completely new to me, I managed to find a way to make it work.

I would like to thank my roommates Rob, Roland and Matthijs for providing support during lockdown, the many coffee breaks and evenings filled with silly TV shows and laughter. Also, I want to thank my study buddy Victor, who was also graduating. He kept me motivated and joined me along the highs and helped me through the lows. I am also very grateful to my good friends Rieneke, Katja and Joram for the many fun walks, talks and dinners during lockdown, and for giving their insight on problems I ran into, whether they were thesis related or not. I would also like to thank my family, Ron and Marie, Emma, Ben and Cécile for always looking after me and showing interest in what I do. Thank you to all the friends that supported me and have proofread my thesis. A big thank you to Umit is in order for inviting me to Prime Vision and being a great colleague and friend throughout the whole project. Finally, I would like to thank my supervisors Ton and Mart for guiding me during the thesis, and giving me the opportunity and freedom to form my research project. Thank you to both for the nice talks we had during coffee, lunch and drinks at the office. Even though I did not always like the thesis life, I really liked the project itself. Thanks to all people mentioned above, I managed to finish my master thesis, a work of which I am proud.

Delft, University of Technology Lucy Smeets
June 28, 2022

Chapter 1

Introduction

1-1 Background

In this modern world, automation is all around us. There are many reasons for processes to be automated, such as the ability of robots to perform tasks that are too dangerous or difficult for humans, achieving higher quality and making less production mistakes, or simply increasing production quantities or process speed. Moreover, machines can work continuously over time, whereas people get tired and need to take breaks. On the other hand, machines are programmed to perform under certain circumstances, and if anything changes, they are less flexible to adapt to the situation. Extensive research and development is focused on how to make automated processes faster, cheaper and more reliable. A general approach to get a better understanding of automated systems is to describe it with a mathematical model and simulate the process.

Automated systems consist of various events, where interactions between these events form the basis of the system dynamics. Such a system is referred to as a discrete event system (DES), where the states have discrete values and state changes are initiated by the occurrence of events, as opposed to time instances, which is the case in discrete time systems [1]. The interpretation of the states is therefore the time instant at which these events happen, and not a value of a physical quantity such as displacement or speed. The coordination of multiple sheets in a printer [2], departures and arrivals of trains [3] or the legs of a robot touching or lifting off the floor [4] are examples of events that define the dynamics of a DES.

There is more that defines a DES than just the occurrence of events. A DES consists of jobs, which are sequences of operations that need to be assigned to the available resources. For example, printing one sheet of paper is a job, consisting of transporting the sheet through the printer and transferring the image to the paper [2]. Finding the optimal assignment of operations to resources is called scheduling. The three main components in scheduling are routing, ordering and synchronization. Routing determines the sequence of resources that a job follows, ordering is needed when multiple jobs need to make use of the same resource and synchronizations specify when a certain operation can start, depending on the status of an operation from another job. Jobs k and $k+1$ are situated in resources R_2 and R_3 in the example system in Figure 1-1. Both jobs choose a route, which is the order in which they enter the resources to get from R_1 to R_4. Since they both want to enter the same resource R_4, an order between the two jobs needs to be chosen, where one job has to wait for the other to finish its operation in R_4. If the operation at resource R_4 is for example an assembly task, and jobs k and $k+1$ contain products that need to be assembled, a synchronization takes place. The operation in R_4 is allowed to start only if the operation from job k in resource R_2 and the operation from job $k+1$ in R_3 are finished.

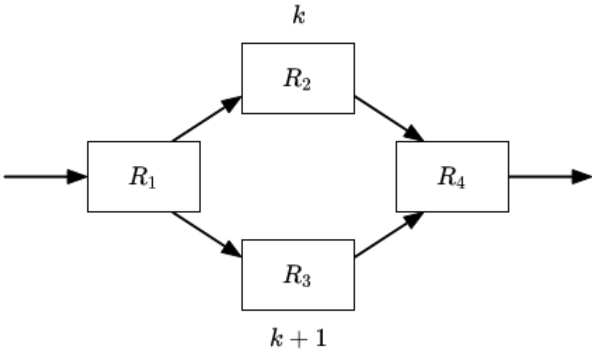


Figure 1-1: Two jobs move through the resources of a DES.

Modeling a DES to achieve an optimal schedule can result in complex, non-linear descriptions in the conventional algebra. However, there is a class of DES that can be described by a model that is linear in the max-plus algebra, which is an algebra that only uses the basic operations maximization and addition. Such models are called max-plus linear (MPL) systems. A great advantage of working with MPL systems is that max-plus linear system theory has a strong analogy to the conventional linear system theory [5, 6].

MPL systems have a fixed system description, in which changes in the structure cannot be modeled. Structure changes are needed when there are multiple routes to take, or orders between operations are variable. Every possible sequence of operations defines a mode, and each mode has its own MPL system description. By introducing a switching element, choosing a different mode for each job is made possible. The resulting system is called a switching max-plus linear (SMPL) system. The examples from before can be extended into SMPL systems by adding the possibility of using different types of paper sheets or simplex and duplex printing [5], switching orders of trains due to disturbances on the network [3], and including multiple gaits that the robot can choose from [4].

Scheduling in SMPL systems has many applications in a wide range of fields, including logistic systems. Meanwhile, the market for automated systems in logistics keeps on growing. Prime Vision is a software company in the sorting industry that works on these types of systems. As global market leader in computer vision integration and robotics for logistics and e-commerce, the company designs innovative solutions to optimize the automation of sorting processes. Their solutions make it easier to scale up the sorting operation, and for operators to sort parcels faster [7]. One of their latest innovations is Autonomous Sorting, where automated guided vehicles form the main element for sorting parcels in warehouse-like environments, such as distribution centers. A test setup is shown in Figure 1-2.

Figure 1-2: Prime Vision robots sorting parcels [8].

In this environment, jobs consist of picking up and dropping off parcels, where the robots can choose between a variety of routes. The situation may occur where robots want to drive on the same location, so they need to wait for each other. Deciding which one can go first asks for the possibility to switch orders. Therefore, the system can be modeled with an SMPL system, where optimal scheduling plays an important role in the efficiency of the system.

1-2 Problem statement

Despite their recent advancements, Prime Vision also continues to conduct research to improve their automated systems. The company aims to both increase the amount of robots and decrease the sorting area in the future. This means that the systems will get more complicated and the robots become harder to control while avoiding conflicting situations like deadlock [9]. Scheduling the robots through the sorting area in a correct and efficient way is hereby of interest. This leads to the main research question for this thesis:

Can the Autonomous Sorting system at Prime Vision be modeled with an SMPL model?

If an SMPL model can be used to describe the sorting system, a scheduling method can be applied to create a proof of concept. Since Prime Vision is flexible in designing floor plans, the model should be flexible as well in order for the model to be of use. The optimal schedules resulting from the model contain information on the navigation on the robots over time. This information can be useful for application to the system. Therefore, three sub questions arise:

1. *Is it possible to create a flexible SMPL model that can be applied to different sorting areas?*
2. *How can an optimal schedule be obtained from the SMPL model?*
3. *How can the resulting optimal schedule be used to control the system?*

After creating the proof of concept, interesting follow-up questions would be whether this method increases the throughput, which is the amount of parcels that gets sorted per hour, and if the optimization is fast enough to execute the computations online.

1-3 Contribution

Existing theory on SMPL systems and model predictive scheduling (MPS) is used to create a versatile model of the sorting system. Modeling the sorting system in chapter 4, an implementation plan of the optimized schedules into the system in chapter 5 and a visualization of the results in chapter 6 are the contributions of this thesis.

1-4 Outline

This thesis is structured as follows. Chapter 2 gives an overview of the sorting process at Prime Vision, where situations that could be improved with scheduling are pointed out. The basics of max-plus algebra and SMPL systems are defined in chapter 3, which are needed to find such optimal schedules. Chapter 4 gives an overview of the equations that describe the dynamics of the sorting system, and a way to turn them into constraints for the optimization problem. Chapter 5 contains theory on a control strategy commonly used for SMPL systems, and the structure and implementation of the optimization problem in Python [10]. The results of the optimization are visualized in chapter 6 and chapter 7 concludes this thesis and gives recommendations for future research.

Chapter 2

Autonomous Sorting

One of the sorting solutions that Prime Vision created is Autonomous Sorting, where the sorting process is made easier and faster by the deployment of robots. The company developed robots that drive autonomously through the sorting area. This chapter describes the workflow of the sorting process. A connection between the sorting process and scheduling is made to illustrate the improvement possibilities of modeling Autonomous Sorting as a switching max-plus linear (SMPL) system.

2-1 Workflow

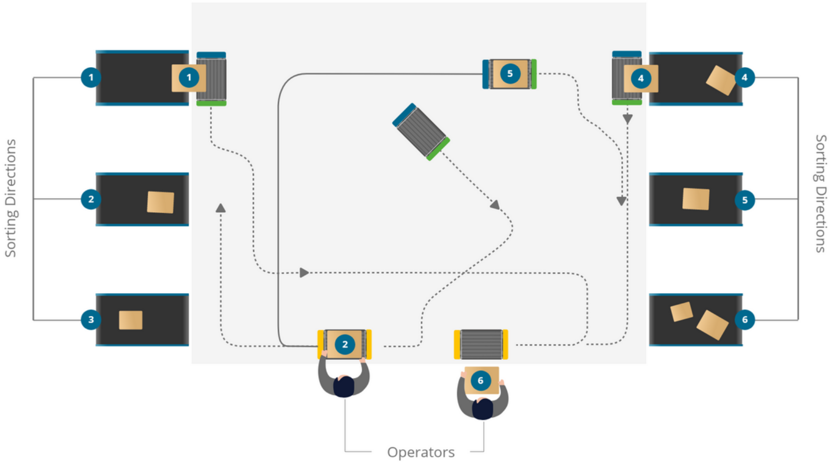


Figure 2-1: Workflow of the sorting process [11].

The sorting system consists of robots, a sorting area where they can drive around, sorting directions where parcels need to be delivered, and inputs where robots can pick up parcels to sort. Operators scan parcels and place them on the robots at the inputs. Figure 2-1 shows an overview of the process. When a parcel is scanned, the robot receives the sorting direction and calculates its path towards it, taking into account planned paths of other robots in the field. For example, the operator on the left scans a parcel that needs to go to sorting direction 2, and the robot plans its path which is depicted by the dotted line. When a robot has delivered the parcel, it finds its way back to an operator to receive the next parcel. In the figure, the robot at sorting direction 1 has planned to drive back to the operator on the right.

2-2 Room for improvement

Each sorting area is described by a floor plan, which can be seen as a graph. A graph consists of nodes and edges, representing locations where the robots can drive to and their connections. An example of a graph representing a random floor plan is shown in Figure 2-2. This graph is designed for the purpose of this thesis and is not being used for real sorting applications. The two green nodes at the bottom are input nodes, where robots can pick up parcels. Some of the blue nodes represent sorting directions where parcels can be dropped off, and others do not have a specific function, other than that they are locations for robots to drive through.

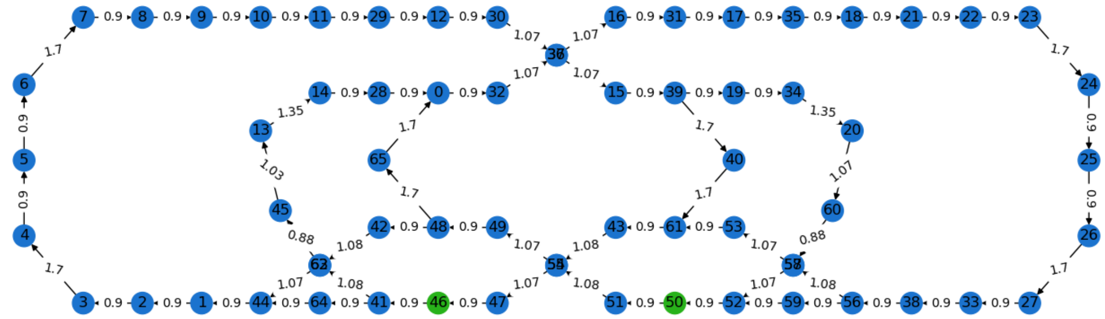
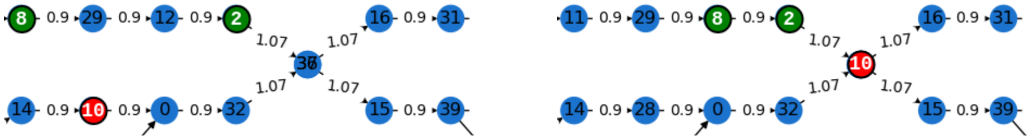


Figure 2-2: Graph representation of a random floor plan.

When a robot picks up a parcel at an input node, it finds its way through the graph to its sorting direction and back to an input node. Details on the current path planning need to stay confidential, but it is important to know that robots can claim a node until they have traveled past it to avoid collisions. The order in which robots currently claim nodes does not always prove to be efficient at crossing or merging paths. In Figure 2-3, an example of a simulation is shown at the crossing at the top of the graph. Even though robot 2 is closer to the crossing and there is enough time for it to go first without robot 10 having to slow down, it waits. This is because robot 10 has claimed a node on the crossing first. The result is that both robots 2 and 8 have to decelerate and wait at the crossing, slowing down the system. These types of situations can be improved on by choosing the optimal order with scheduling.

Since the path planning is based on weighted, directed graphs, it allows for modeling the system as an SMPL system, which will become clear in chapter 3. By computing optimal choices in these kinds of situations, the flow of robots through the sorting area would be increased, which in turn increases the throughput of the system.



(a) The crossing is currently not occupied.

(b) Robots 2 and 8 wait for robot 10 at the crossing.

Figure 2-3: An example of inefficient ordering at a crossing. Robot 2 could cross before robot 10 without raising any conflicts, but robot 10 claimed the nodes on the crossing first.

2-3 Autonomous sorting as a scheduling problem

The sorting process can be described in terms of events, where the event counter k resembles the k^{th} time a robot receives a parcel to sort. In this system, k can be referred to as an event, job or robot, since each robot k has one job. This job consists of a sequence of operations on the available resources, which are the nodes in the graph. The operations in a job consist of picking up and dropping off a parcel, and arriving at nodes. In this description, two assumptions are made. First of all, robots cannot overtake other robots between two consecutive nodes. Secondly, each node can be occupied by only one robot at a time.

As previously mentioned, the states in a discrete event system (DES) resemble time instants at which events happen. The states in this system are denoted by $x_i(k)$, which is the time at which robot k enters node i. The states of robot k in all nodes in the graph therefore resembles the route that robot k travels, along with a schedule on when it arrives in each node. The route description for robot k also depends on the travel time between any two consecutive nodes i and j, denoted by $\tau_{i,j}(k)$, and on the moment that an operator scans a parcel. This moment can be seen as an external input signal, denoted by $u_e(k)$.

All elements needed to describe the movements of one robot through the graph have been defined. The route description is a set of constraints for each of the nodes visited by the robot. Say that robot k receives a parcel at node 46. The robot can enter the next node on the path after the parcel became available, and after robot k entered node 46. This leads to the following constraints:

$$\begin{aligned} x_{41}(k) &\geq u_e(k) + \tau_{46,41}(k) \\ x_{41}(k) &\geq x_{46}(k) + \tau_{46,41}(k) \end{aligned} \tag{2-1}$$

Since both constraints need to be respected, they can be combined and simplified into one constraint:

$$x_{41}(k) \geq \max\left(u_e(k),\ x_{46}(k)\right) + \tau_{46,41}(k) \tag{2-2}$$

Such a constraint can be obtained for each node on the path of robot k. Note that the constraints only consist of the operations maximization and addition. Traveling a route can therefore be modeled by a max-plus linear (MPL) system. In most floor plans, multiple routes are available. Each route has its own system description. Every time a robot starts a new job, it can choose between these routes. Enabling and disabling routes is made possible by introducing a switching element, which is characteristic for SMPL systems.

Another characteristic of SMPL systems is the switching or orders of different jobs on the same or different resources. This is the case when multiple robots are driving through the graph. When robots meet each other at a crossing, a choice needs to be made for the order in which the robots may cross. This introduces a different type of constraints, where the state of robot k depends on the states of other robots $k \pm \mu$, where $\mu = 1, \dots, \mu_{\max}$. The different types of constraints are discussed in more detail in chapter 3 and 4.

Chapter 3

Switching max-plus linear systems

Making a model of a discrete event system (DES) can result in complex, non-linear system descriptions. However, max-plus linear (MPL) systems that describe a DES are linear in the max-plus algebra. This algebra has maximization and addition as its basic operators, and has a strong analogy to the conventional algebra. This chapter gives an introduction to max-plus algebra and MPL systems. A general framework for constructing switching max-plus linear (SMPL) systems is shown, which touches upon the three main elements in scheduling: routing, ordering and synchronization. The information in this chapter is based on [5, 6, 12].

3-1 Introduction to max-plus algebra

This section discusses the basics of the max-plus algebra, which are needed to understand the differences with working in the conventional algebra. The max-plus algebra defines different operations for scalars and matrices.

3-1-1 Basic definitions and operations

The max-plus algebra is defined by the structure $(\mathbb{R}_\varepsilon, \oplus, \otimes)$. Here, the set $\mathbb{R}_\varepsilon$ consists of $\mathbb{R} \cup \{\varepsilon\}$, with $\varepsilon = -\infty$. The operator $\oplus$ denotes a maximization, and $\otimes$ an addition. In Equation 3-1, the definition of these operators is shown for $x, y \in \mathbb{R}_\varepsilon$.

$$\begin{aligned} x \oplus y &= \max(x, y) \\ x \otimes y &= x + y \end{aligned} \tag{3-1}$$

The appearance of these symbols is chosen to emphasize the analogy between both the max-plus addition $\oplus$ and conventional addition, and the max-plus multiplication $\otimes$ and conventional multiplication. By substituting + with $\oplus$ and × with $\otimes$, many properties from conventional linear algebra still hold in the max-plus algebra. One important example is the distributive property of $\otimes$ over $\oplus$, as $\times$ also has priority over $+$. Associativity and commutativity for $\oplus$ and $\otimes$ also hold. Note that ε acts as the neutral element for $\oplus$, since $a \oplus \varepsilon = a = \varepsilon \oplus a, \forall a \in \mathbb{R}_\varepsilon$. In the same way, 0 acts as the neutral element for $\otimes$, as $a \otimes 0 = a = 0 \otimes a, \forall a \in \mathbb{R}_\varepsilon$.

For $r, x \in \mathbb{R}$, $x^{\otimes^r}$ is the r^{th} max-plus algebraic power of x and it translates to rx in conventional algebra. Therefore, $x^{\otimes^0} = 0$, and by definition also $\varepsilon^{\otimes^0} = 0$. When $r > 0$, $\varepsilon^{\otimes^r} = \varepsilon$, and for $r < 0$ the max-plus algebraic power of ε is not defined.

Lastly, the max-plus binary variable w is defined. Let $w \in \mathbb{B}_\varepsilon = \{0, \varepsilon\}$, then its adjoint $\bar{w} \in \mathbb{B}_\varepsilon$ is defined as follows:

$$\bar{w} = \begin{cases} 0 & \text{if } w = \varepsilon \\ \varepsilon & \text{if } w = 0 \end{cases} \tag{3-2}$$

Max-plus binary variables and their adjoint are related to conventional binary variables, which are denoted by $w^\flat$, as follows:

$$\begin{aligned} w &= \beta w^\flat \\ \bar{w} &= \beta(1 - w^\flat) \end{aligned} \tag{3-3}$$

Here, β is a large negative number, for example -10.000. The max-plus binary variables are related to the conventional binary variables as follows:

$$w^\flat = \begin{cases} 0 & \text{if } w = 0 \\ 1 & \text{if } w = \varepsilon \end{cases} \tag{3-4}$$

3-1-2 Matrix definitions and operations

For matrices $A, B \in \mathbb{R}_\varepsilon^{m \times n}$ and $C \in \mathbb{R}_\varepsilon^{n \times p}$, the basic max-plus algebraic matrix operations are defined as follows for all i, j.

$$\begin{aligned} [A \oplus B]_{ij} &= [A]_{ij} \oplus [B]_{ij} = \max([A]_{ij}, [B]_{ij}) \\ [A \otimes C]_{ij} &= \bigoplus_{k=1}^{n} [A]_{ik} \otimes [C]_{kj} = \max_{k=1,\ldots,n}([A]_{ik} + [C]_{kj}) \\ [A \odot B]_{ij} &= [A]_{ij} + [B]_{ij} \end{aligned} \tag{3-5}$$

The bottom equation defines the max-plus Schur product, denoted by $\odot$. The max-plus algebraic zero matrix $\mathcal{E}_{m \times n} \in \mathbb{R}_\varepsilon^{m \times n}$ is defined as $[\mathcal{E}_{m \times n}]_{ij} = \varepsilon$ for all i, j. The max-plus algebraic identity matrix $E_n \in \mathbb{R}_\varepsilon^{n \times n}$ is defined as $[E_n]_{ii} = 0, \forall i$, and $[E_n]_{ij} = \varepsilon, \forall i \neq j$. Examples of max-plus algebraic matrix operations and the zero and identity matrices are shown in Equation 3-6 and Equation 3-7.

$$\begin{aligned}
\begin{bmatrix}2 & 0\\ \varepsilon & 5\end{bmatrix} \oplus \begin{bmatrix}3 & 1\\ 6 & \varepsilon\end{bmatrix} &= \begin{bmatrix}2\oplus 3 & 0\oplus 1\\ \varepsilon\oplus 6 & 5\oplus\varepsilon\end{bmatrix} = \begin{bmatrix}\max(2,3) & \max(0,1)\\ \max(\varepsilon,6) & \max(5,\varepsilon)\end{bmatrix} = \begin{bmatrix}3 & 1\\ 6 & 5\end{bmatrix}\\
\begin{bmatrix}2 & 0\\ \varepsilon & 5\end{bmatrix} \otimes \begin{bmatrix}3 & 1\\ 6 & \varepsilon\end{bmatrix} &= \begin{bmatrix}(2\otimes 3)\oplus(0\otimes 6) & (2\otimes 1)\oplus(0\otimes\varepsilon)\\ (\varepsilon\otimes 3)\oplus(5\otimes 6) & (\varepsilon\otimes 1)\oplus(5\otimes\varepsilon)\end{bmatrix}\\
&= \begin{bmatrix}\max(2+3,0+6) & \max(2+1,0+\varepsilon)\\ \max(\varepsilon+3,5+6) & \max(\varepsilon+1,5+\varepsilon)\end{bmatrix} = \begin{bmatrix}6 & 3\\ 11 & \varepsilon\end{bmatrix}\\
\begin{bmatrix}2 & 0\\ \varepsilon & 5\end{bmatrix} \odot \begin{bmatrix}3 & 1\\ 6 & \varepsilon\end{bmatrix} &= \begin{bmatrix}2+3 & 0+1\\ \varepsilon+6 & 5+\varepsilon\end{bmatrix} = \begin{bmatrix}5 & 1\\ \varepsilon & \varepsilon\end{bmatrix}
\end{aligned} \tag{3-6}$$

$$\mathcal{E}_{m\times n} = \begin{bmatrix}\varepsilon & \varepsilon & \dots & \varepsilon\\ \varepsilon & \varepsilon & & \vdots\\ \vdots & & \ddots & \varepsilon\\ \varepsilon & \dots & \varepsilon & \varepsilon\end{bmatrix} \qquad E_n = \begin{bmatrix}0 & \varepsilon & \dots & \varepsilon\\ \varepsilon & 0 & & \vdots\\ \vdots & & \ddots & \varepsilon\\ \varepsilon & \dots & \varepsilon & 0\end{bmatrix} \tag{3-7}$$

The max-plus algebraic power of a matrix $A \in \mathbb{R}_\varepsilon^{n\times n}$ is defined as $A^{\otimes^k} = A \otimes A^{\otimes^{(k-1)}}$ for $k = 1, 2, \dots$, where $A^{\otimes^0} = E_n$. The inverse of a matrix $S \in \mathbb{R}_\varepsilon^{n\times n}$ in max-plus algebra is denoted by $S^{\otimes^{-1}}$, where $S^{\otimes^{-1}} \otimes S = S \otimes S^{\otimes^{-1}} = E_n$. A matrix S is only max-plus invertible if it has precisely one entry in each row and column different from ε, so S should be diagonal up to a permutation of rows. Therefore we can write it as a max-plus diagonal matrix $S = \text{diag}_\oplus(s_1, \dots, s_n)$, where the diagonal elements are $s_1, \dots, s_n$ and all other elements are ε. Its inverse is defined as $S^{\otimes^{-1}} = \text{diag}_\oplus(-s_1, \dots, -s_n)$.

Finally, the max-plus Kleene star of a matrix is defined as $A^* = \bigoplus_{k=0}^{\infty} A^{\otimes^k}$. It exists for any square matrix A, with only non-positive circuit weights in its precedence graph $\mathcal{G}(A)$. These terms are explained in section 3-2. The max-plus Kleene star is needed to solve a max-plus linear equation of the form $x = A \otimes x \oplus b$ with $x, b \in \mathbb{R}_\varepsilon^n$. In Equation 3-8, the solution to this type of equation is shown. This is important for solving max-plus linear state space models, as will be shown in section 3-3.

$$x = A^* \otimes b \tag{3-8}$$

Another form of max-plus linear equations is $A \otimes x = b$, with $A \in \mathbb{R}_\varepsilon^{m\times n}$, $x \in \mathbb{R}_\varepsilon^n$ and $b \in \mathbb{R}_\varepsilon^m$. This type does not always have a solution, but the largest subsolution can always be computed. We call x a subsolution of the system of max-plus linear equations $A \otimes x = b$ if $A \otimes x \leq b$. The largest subsolution is denoted by $x^*(A, b)$ and defined as

$$[x^*(A,b)]_j = \min_{i\in\{1,\dots,m\}} (b_i - a_{ij}), \quad \text{for } j = 1, \dots, n \tag{3-9}$$

3-2 Graphs and max-plus algebra

Any square matrix $A \in \mathbb{R}_{\varepsilon}^{n \times n}$ has a precedence graph $\mathcal{G}(A)$, which is a weighted, directed graph. It has a finite set of nodes and edges, denoted by $\mathcal{N}(A)$ and $\mathcal{D}(A) \subset \mathcal{N}(A) \times \mathcal{N}(A)$ respectively. An edge $(i, j) \in \mathcal{D}(A)$ is an outgoing edge at node i and incoming at j. The weight of this edge is denoted by τ_{ij}, which represents the time it takes to go from node i to node j. The weights are elements of the matrix A:

$$A_{ji} = \begin{cases} \tau_{ij} & \text{if there exists an edge from node } i \text{ to node } j\text{: } (i,j) \in \mathcal{D}(A) \\ \varepsilon & \text{otherwise} \end{cases} \tag{3-10}$$

A path $p = \{(i_k, j_k)\}$, for $k = 1, \ldots, m$ from node i to j is a sequence of m edges, such that $(i_k, j_k) \in \mathcal{D}(A)$, $\forall k \in \{1, \ldots, m\}$, where $i_1 = i$, $j_m = j$ and $i_{k+1} = j_k$ for $k < m$. The length $|p|_l$ of path p is the amount of edges in the path, which is m. The weight $|p|_w$ is the sum of the weights on the path, which is calculated as follows:

$$|p|_w = \bigotimes_{k=1}^{m} \tau_{i_k j_k} \tag{3-11}$$

When a path ends in the starting node, so when $i = j$, it is called a circuit. The length and weight of a circuit are determined in the same way as for a regular path.

The max-plus algebraic power of matrix $A \in \mathbb{R}_{\varepsilon}^{n \times n}$ has a certain practical meaning associated to the precedence graph $\mathcal{G}(A)$. For $k \in \mathbb{N} \setminus \{0\}$, $(A^{\otimes^k})_{ij}$ is the maximal weight of all paths in $\mathcal{G}(A)$ from node j to i of length k, which is computed by

$$(A^{\otimes^k})_{ij} = \max_{i_1, i_2, \ldots, i_{k-1} \in \mathcal{N}(A)} (a_{ii_1} + a_{i_1 i_2} + \cdots + a_{i_{k-1} j}) \tag{3-12}$$

If no paths of length k from j to i exist, the maximal weight is defined as ε.

A directed graph $\mathcal{G}$ is strongly connected if there exists a path from i to j for any two different nodes $i, j \in \mathcal{N}$. A matrix $A \in \mathbb{R}_{\varepsilon}^{n \times n}$ is irreducible if its precedence graph $\mathcal{G}(A)$ is strongly connected. This translates to the following mathematical definition: a matrix $A \in \mathbb{R}_{\varepsilon}^{n \times n}$ is irreducible if

$$\left(\bigoplus_{k=1}^{n-1} A^{\otimes^k} \right)_{ij} \neq \varepsilon \qquad \text{for all } i, j \text{ with } i \neq j \tag{3-13}$$

Indeed, this shows that for any two different nodes i and j of $\mathcal{G}(A)$, there exists at least one path from j to i of length $1, 2, \ldots$ or $n - 1$.

3-3 Max-plus linear systems

When working with max-plus linear discrete event systems, the states do not get evaluated at time instances, but at each event, where the event counter is denoted by k. The MPL system for a DES is defined in Equation 3-14.

$$\begin{cases} x(k) = A \otimes x(k-1) \oplus B \otimes u(k) \\ y(k) = C \otimes x(k) \end{cases} \tag{3-14}$$

Here, $A \in \mathbb{R}_{\varepsilon}^{n \times n}$, $B \in \mathbb{R}_{\varepsilon}^{n \times n_u}$ and $C \in \mathbb{R}_{\varepsilon}^{n_y \times n}$ are the system matrices, with n the number of states, n_u the number of inputs and n_y the number of outputs. The vector $x(k) \in \mathbb{R}_{\varepsilon}^{n}$ is the state vector, $u(k) \in \mathbb{R}_{\varepsilon}^{n_u}$ is the input vector and $y(k) \in \mathbb{R}_{\varepsilon}^{n_y}$ the output vector. The values of the states, inputs and outputs all represent event times.

3-3-1 Linearity

The linearity of the system can be shown by first studying the input-output behaviour. By Equation 3-14, the following holds:

$$\begin{aligned} x(1) &= A \otimes x(0) \oplus B \otimes u(1) \\ x(2) &= A \otimes x(1) \oplus B \otimes u(2) \\ &= A^{\otimes^2} \otimes x(0) \oplus A \otimes B \otimes u(1) \oplus B \otimes u(2) \\ &\vdots \\ x(k) &= A^{\otimes^k} \otimes x(0) \oplus \bigoplus_{i=1}^{k} A^{\otimes^{k-i}} \otimes B \otimes u(i) \end{aligned} \tag{3-15}$$

The bottom equation in Equation 3-15 gives the general state event times for any event k in the future. Substituting this expression into the output equation from Equation 3-14 yields

$$y(k) = C \otimes A^{\otimes^k} \otimes x(0) \oplus \bigoplus_{i=1}^{k} C \otimes A^{\otimes^{k-i}} \otimes B \otimes u(i) \tag{3-16}$$

Define two input sequences $u_1 = \{u_1(k)\}_{k=1}^{\infty}$ and $u_2 = \{u_2(k)\}_{k=1}^{\infty}$ and let $y_1 = \{y_1(k)\}_{k=1}^{\infty}$ be the output sequence corresponding to the input sequence u_1 and $y_2 = \{y_2(k)\}_{k=1}^{\infty}$ to input sequence u_2, with initial conditions $x_1(0) = x_{1,0}$ and $x_2(0) = x_{2,0}$. Now, the linearity of the MPL system is shown by substituting the input sequence $\alpha \otimes u_1 \oplus \beta \otimes u_2$ with initial condition $\alpha \otimes x_{1,0} \oplus \beta \otimes x_{2,0}$ into the expression from Equation 3-16, resulting in the output sequence $\alpha \otimes y_1 \oplus \beta \otimes y_2$ [13].

3-3-2 Solving max-plus linear systems

Often in max-plus linear systems, the output is the same as the states, so $y(k) = x(k)$. An explicit state equation for the description of the system remains:

$$x(k) = A \otimes x(k-1) \oplus B \otimes u(k) \tag{3-17}$$

As will be shown in subsection 3-4-1, the states often do not only depend on state values from previous events, but also on states from the same event k. This results in an implicit model of the form

$$x(k) = A_0 \otimes x(k) \oplus A_1 \otimes x(k-1) \oplus B_0 \otimes u(k) \tag{3-18}$$

Remember from Equation 3-8 in section 3-1 that the solution to an equation of the form $x = A \otimes x \oplus b$ is $x = A^* \otimes b$. Define $b = A_1 \otimes x(k-1) \oplus B_0 \otimes u(k)$, then the explicit solution to Equation 3-18 is as given in Equation 3-19, if A_0^* exists. The max-plus Kleene star of any square matrix exists if its precedence graph has only non-positive circuit weights.

$$x(k) = \underbrace{A_0^* \otimes A_1}_{A} \otimes x(k-1) \oplus \underbrace{A_0^* \otimes B_0}_{B} \otimes u(k) \tag{3-19}$$

In max-plus linear models, states from event k can depend on states from both the same and the previous event, as shown before. It is also possible that states depend on earlier events $k-\mu$, where μ can go up until a certain number $\mu_{\max}$. This results in an implicit model of the following form:

$$x(k) = \bigoplus_{\mu=0}^{\mu_{\max}} \left(\tilde{A}_\mu \otimes x(k-\mu)\right) \oplus B_0 \otimes u(k) \tag{3-20}$$

To solve for $x(k)$, define $b = \bigoplus_{\mu=1}^{\mu_{\max}} \left(\tilde{A}_\mu \otimes x(k-\mu)\right) \oplus B_0 \otimes u(k)$. Then if $\tilde{A}_0^*$ exists:

$$x(k) = \bigoplus_{\mu=1}^{\mu_{\max}} \left(\underbrace{\tilde{A}_0^* \otimes \tilde{A}_\mu}_{A_\mu} \otimes x(k-\mu)\right) \oplus \underbrace{\tilde{A}_0^* \otimes B_0}_{B} \otimes u(k) \tag{3-21}$$

3-4 Switching max-plus linear systems

Some discrete event systems might have multiple ways for jobs to be executed on the available resources. Each of these job routes has a different structure and therefore results in a different system description. The dynamics of such a system can not be captured in one MPL system. Each possible route and order of events is called a mode of the system and can be modeled by its own MPL system. As a result, each event k can choose from the available modes and the system matrices become dependent on k. Max-plus binary variables are introduced which enable the possibility to switch between modes; they enable or disable certain modes. The complete system containing all max-plus binary variables is called a switching max-plus linear system.

In scheduling, three aspects are important. Routing is used to determine the sequence of resources for the operations of a job, ordering determines the order of operations of different jobs on one resource and synchronization handles situations where an operation depends on an operation from another job, and on a different resource. In this section, a general framework to model all three scheduling aspects is discussed for SMPL systems, where the max-plus binary variables resemble choices for routes, orders and synchronizations.

3-4-1 Routing

Each job consists of multiple operations, which take place on resources. The order in which these resources are visited defines the route of the job. The solution to the system equations for a route defines the time instants that each operation can start on its assigned resource, depending on the starting and processing time of the previous operation on the preceding resource.

Consider a system that has one job per cycle k, consisting of p_k operations, that each take place on a resource $r \in \{1, \ldots, n\}$. Only the systems with at least as many resources as operations are considered, where each resource can only be visited once per job. Define the sequence of used resources as $(r_1, \ldots, r_{p_k})$. Let $\tau_{r_j}(k) \geq 0$ be the time it takes to execute operation j on resource r_j for job k for $j = 1, \ldots, p_k$, also called the processing time. Let $x(k) = [x_1(k) \quad \ldots \quad x_n(k)]^T$ be the vector of starting times on all resources for job k, where a state $x_r(k) \geq 0$ if resource r is visited by job k, and $x_r(k) = \varepsilon$ if not. This routing structure raises the following set of inequalities, which ensure that an operation on resource r_i can only start after it has finished its previous operation j on resource r_j, with $i > j$ and $i, j \in \{1, \ldots, p_k\}$:

$$x_{r_i}(k) \geq x_{r_j}(k) + \tau_{r_j}(k) \tag{3-22}$$

Equation 3-22 is not an equality, since ordering and synchronizations could delay starting times. Note however, that the operations can not start sooner than when the previous operation has finished. For the simple case where each resource is visited in ascending order, the inequalities can be combined into max-plus matrix notation as follows:

$$\begin{bmatrix} x_1(k) \\ x_2(k) \\ \vdots \\ x_{n-1}(k) \\ x_n(k) \end{bmatrix} \geq \begin{bmatrix} \varepsilon & \ldots & \ldots & \ldots & \varepsilon \\ \tau_{r_1}(k) & \varepsilon & \ldots & \ldots & \varepsilon \\ \varepsilon & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \varepsilon & \ldots & \varepsilon & \tau_{r_{n-1}}(k) & \varepsilon \end{bmatrix} \otimes \begin{bmatrix} x_1(k) \\ x_2(k) \\ \vdots \\ x_{n-1}(k) \\ x_n(k) \end{bmatrix} \tag{3-23}$$

This can be written compactly as $x(k) \geq A^{\text{route}}(k) \otimes x(k)$. But in practice, a job does not always use all available resources, let alone visit them in the same, fixed order. Each job k can choose its own route, and the presence of such a choice is what distinguishes an SMPL system from an MPL system. Choices can be made by introducing max-plus binary variables.

Let L be the number of alternative routes, and let $w_\ell(k) \in \mathbb{B}_\varepsilon$ for $\ell = 1, \ldots, L$ be the variables that indicate whether route ℓ is used in job k or not:

$$w_\ell(k) = \begin{cases} 0 & \text{if route } \ell \text{ is used in job } k \\ \varepsilon & \text{if route } \ell \text{ is not used in job } k \end{cases} \tag{3-24}$$

Note that only one route can be chosen for each job. If job k uses route ℓ, then $w_\ell(k) = 0$ and all other routing variables are forced to be $w_l(k) = \varepsilon$ for all $l \neq \ell$.

Each route consists of a path through the resources. As discussed in section 3-2, an SMPL system has a precedence graph, in which the resources are represented by the nodes and the edges between nodes denote the possible transitions between resources. Each route ℓ visits a sequence of nodes, using certain edges in the precedence graph. Let route ℓ consist of edges (r_i, r_j), where $i \in \{1, \dots, p_k - 1\}$, $j = i + 1$ and $r_i, r_j \in \{1, \dots, n\}$. Let $s_{r_i,r_j}(k) \in \mathbb{B}_\varepsilon$ be the variables that denote whether edge (r_i, r_j) is used by job k:

$$s_{r_i,r_j}(k) = \begin{cases} 0 & \text{if edge } (r_i, r_j) \text{ is used in job } k \\ \varepsilon & \text{if edge } (r_i, r_j) \text{ is not used in job } k \end{cases} \tag{3-25}$$

Each route has a unique set of active edge variables. Whether an edge is active or not is determined by the following relation, where $L_{i,j}$ is the set of routes that include edge (r_i, r_j):

$$s_{r_i,r_j}(k) = \bigoplus_{l \in L_{i,j}} w_l(k) \tag{3-26}$$

If the chosen route for job k is in that set, $s_{r_i,r_j}(k)$ will be active, and when none of the routes in the set are chosen, $s_{r_i,r_j}(k)$ will be inactive.

Each route has a different set of edges, and therefore has its own system description, resulting in the system matrix $A^{\text{route}}(w(k))$, where $w(k) = [w_1(k) \quad \dots \quad w_L(k)]^T$. The general structure of the system matrix is shown in Equation 3-27, where the job index k is left out for clarity. Note that since each resource can only be visited once, each row has at most one element that is different from ε after substituting the values of the edge decision variables.

$$\begin{bmatrix} x_1(k) \\ x_2(k) \\ \vdots \\ x_{n-1}(k) \\ x_n(k) \end{bmatrix} \geq \begin{bmatrix} \varepsilon & \tau_2 \otimes s_{2,1} & \cdots & \cdots & \tau_n \otimes s_{n,1} \\ \tau_1 \otimes s_{1,2} & \varepsilon & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \tau_n \otimes s_{n,n-1} \\ \tau_1 \otimes s_{1,n} & \cdots & \cdots & \tau_{n-1} \otimes s_{n-1,n} & \varepsilon \end{bmatrix} \otimes \begin{bmatrix} x_1(k) \\ x_2(k) \\ \vdots \\ x_{n-1}(k) \\ x_n(k) \end{bmatrix} \tag{3-27}$$

By substituting the values for $s_{r_i,r_j}(k)$ into this matrix, which follow directly from the chosen route $w_\ell(k)$ through Equation 3-26, the system matrix for route ℓ is obtained, denoted by $A^{\text{route}}_\ell(k)$. Finally, the routing system matrix for job k can be obtained through

$$A^{\text{route}}(w(k)) = \bigoplus_{\ell=1}^{L} w_\ell(k) \otimes A^{\text{route}}_\ell(k) \tag{3-28}$$

3-4-2 Ordering

In practice, multiple jobs take place in a DES at the same time, and it is likely that operations from different jobs are assigned to the same resource. Since a resource can only be used by one operation at a time, the order of operations on that resource needs to be determined. A distinction is made between nodes in the precedence graph that have one incoming edge and nodes with multiple incoming edges. Due to the restriction that jobs can not overtake one another on edges, the order of operations on nodes with one incoming edge is fixed. If two jobs use the same (part of a) route, the job that started first on that route also enters all nodes on the common part of the route first.

Let $\mathcal{N}_{\text{ord}}$ be the set of nodes with multiple incoming edges. To be able to choose the order of operations at such a resource, a new max-plus binary variable is needed that takes into account other jobs. These could be jobs that started earlier or later than job k, therefore let $\pm\mu = 1, \ldots, \mu_{\text{max}}$. Setting a negative value for μ allows for switching orders with future jobs. Let $z_{r,\mu}(k-\mu) \in \mathbb{B}_\varepsilon$ be the ordering variable for all $r \in \mathcal{N}_{\text{ord}}$, defined as

$$z_{r,\mu}(k-\mu) = \begin{cases} 0 & \text{if the operation from job } k-\mu \text{ in resource } r \text{ precedes} \\ & \text{the operation from job } k \text{ in resource } r \\ \varepsilon & \text{otherwise} \end{cases} \tag{3-29}$$

The general max-plus linear state equations for ordering are given by

$$x_r(k) \geq x_r(k-\mu) \otimes \tau_r(k-\mu) \otimes z_{r,\mu}(k-\mu) \tag{3-30a}$$
$$x_r(k) \geq x_r(k+\mu) \otimes \tau_r(k+\mu) \otimes \bar{z}_{r,\mu}(k) \tag{3-30b}$$

These inequalities only define time constraints for job k, given that it has to wait for job $k+\mu$ or $k-\mu$ in resource r. If the choice is made for job k to go first, the following inequalities are needed:

$$x_r(k-\mu) \geq x_r(k) \otimes \tau_r(k) \otimes \bar{z}_{r,\mu}(k-\mu) \tag{3-30c}$$
$$x_r(k+\mu) \geq x_r(k) \otimes \tau_r(k) \otimes z_{r,\mu}(k) \tag{3-30d}$$

Provided that both job k and job $k-\mu$ use resource r, exactly one of the inequalities in Equation 3-30a and 3-30c holds. The other becomes negligible since it requires the state to be larger than ε, which is always true. The same holds for k and $k+\mu$ in Equation 3-30b and 3-30d.

For $\mu < 0$ define $\mu^+ = -\mu$ and note that the ordering variable becomes $z_{r,-\mu^+}(k+\mu^+)$, which defines the relation between job k and a future job $k+\mu^+$ in each resource with multiple incoming edges. Furthermore, $z_{r,-\mu^+}(k+\mu^+)$ is defined as $\bar{z}_{r,\mu^+}(k)$.

Let $Z = |\mathcal{N}_{\text{ord}}|$ be the number of nodes with multiple incoming edges and let $z_\mu(k-\mu)$ be a vector of length Z containing the ordering variables between jobs k and $k-\mu$ on all resources with multiple incoming edges, such that for $\zeta \in \{1, \ldots, Z\}$, $[z_\mu(k-\mu)]_\zeta = z_{r_\zeta,\mu}(k-\mu)$. The ordering constraints for job k from Equation 3-30a and 3-30b can be captured in system matrices $A_\mu^{\text{ord}}(z_\mu(k-\mu))$ for all $\pm\mu = 1, \ldots, \mu_{\text{max}}$. The state equations for ordering are given by Equation 3-31, where the general structure of $A_\mu^{\text{ord}}(z_\mu(k-\mu))$ is shown, in the case that every node has multiple incoming edges. The job indices $(k-\mu)$ for both the processing times and the ordering variables are left out for clarity.

$$\begin{bmatrix} x_1(k) \\ x_2(k) \\ \vdots \\ x_n(k) \end{bmatrix} \geq \begin{bmatrix} \tau_1 \otimes z_{1,\mu} & \varepsilon & \dots & \varepsilon \\ \varepsilon & \tau_2 \otimes z_{2,\mu} & & \vdots \\ \vdots & & \ddots & \varepsilon \\ \varepsilon & \dots & \varepsilon & \tau_n \otimes z_{n,\mu} \end{bmatrix} \otimes \begin{bmatrix} x_1(k-\mu) \\ x_2(k-\mu) \\ \vdots \\ x_n(k-\mu) \end{bmatrix} \tag{3-31}$$

Compactly written, the system considering all μ becomes

$$x(k) \geq \bigoplus_{\pm\mu=1}^{\mu_{\max}} A_{\mu}^{\text{ord}}(z_\mu(k-\mu)) \otimes x(k-\mu) \tag{3-32}$$

Each node $r_\zeta \in \mathcal{N}_{\text{ord}}$ provides a separate system matrix $A_{\mu,\zeta}^{\text{ord}}(k-\mu)$ with only finite elements on the diagonal, given by

$$[A_{\mu,\zeta}^{\text{ord}}(k-\mu)]_{rr} = \begin{cases} \tau_r & \text{if job } k \text{ has to be scheduled on resource } r \text{ after job } k-\mu \\ \varepsilon & \text{otherwise} \end{cases} \tag{3-33}$$

The ordering system matrix for job k with respect to $k-\mu$ is obtained by Equation 3-34, which is a unique matrix for every feasible combination of orders between any two jobs.

$$A_{\mu}^{\text{ord}}(z_\mu(k-\mu)) = \bigoplus_{\zeta=1}^{Z} [z_\mu(k-\mu)]_\zeta \otimes A_{\mu,\zeta}^{\text{ord}}(k-\mu) \tag{3-34}$$

3-4-3 Synchronization

A situation where operations from different jobs have to wait for each other, in which they are assigned to different resources, is called a synchronization. The choice is to be made which of the operations can be executed first. Synchronizations therefore have many similarities with ordering.

Let L_{syn} be the set of synchronization modes in which each mode is coupled to two resources, and let $b_{\ell,\mu}(k-\mu) \in \mathbb{B}_\varepsilon$ be the synchronization variable for all $\ell \in L_{\text{syn}}$, defined as

$$b_{\ell,\mu}(k-\mu) = \begin{cases} 0 & \text{if synchronization } \ell \text{ is made where job } k \text{ waits for } k-\mu \\ \varepsilon & \text{otherwise} \end{cases} \tag{3-35}$$

The general max-plus linear state equations for synchronization are given by

$$x_i(k) \geq x_r(k-\mu) \otimes \tau_r(k-\mu) \otimes b_{\ell,\mu}(k-\mu) \tag{3-36a}$$

$$x_i(k) \geq x_r(k+\mu) \otimes \tau_r(k+\mu) \otimes \bar{b}_{\ell,\mu}(k) \tag{3-36b}$$

They are defined for any combination of resources $i, r \in \{1, \dots, n\}$ that requires a synchronization. These inequalities only define time constraints for job k, given that k has to wait for job $k + \mu$ or $k - \mu$. If the choice is made for job k to go first, the following inequalities are needed instead:

$$x_r(k - \mu) \geq x_i(k) \otimes \tau_i(k) \otimes \bar{b}_{\ell,\mu}(k - \mu) \tag{3-36c}$$

$$x_r(k + \mu) \geq x_i(k) \otimes \tau_i(k) \otimes b_{\ell,\mu}(k) \tag{3-36d}$$

Let $S = |L_{\text{syn}}|$ be the number of nodes that need synchronization and let $b_\mu(k-\mu)$ be a vector of length S containing all synchronization variables, such that for $\sigma \in \{1, \dots, S\}$, $[b_\mu(k-\mu)]_\sigma = b_{\ell_\sigma,\mu}(k - \mu)$. The synchronization constraints for job k from Equation 3-36a and 3-36b can be summarized in system matrices $A^{\text{syn}}_\mu(b_\mu(k - \mu))$ for all $\pm\mu = 1, \dots, \mu_{\max}$. The state equations for synchronization are given in Equation 3-37 with the most general structure of $A^{\text{syn}}_\mu(b_\mu(k-\mu))$. The job indices $(k-\mu)$ for both the processing times and the synchronization variables are left out for clarity.

$$\begin{bmatrix} x_1(k) \\ x_2(k) \\ \vdots \\ x_{n-1}(k) \\ x_n(k) \end{bmatrix} \geq \begin{bmatrix} \varepsilon & \tau_2 \otimes b_{\ell_1,\mu} & \cdots & \cdots & \tau_n \otimes b_{\ell_{n-1},\mu} \\ \tau_1 \otimes b_{\ell_n,\mu} & \varepsilon & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \tau_n \otimes b_{\ell_{S-n-1},\mu} \\ \tau_1 \otimes b_{\ell_{S-n-2},\mu} & \cdots & \cdots & \tau_{n-1} \otimes b_{\ell_S,\mu} & \varepsilon \end{bmatrix} \otimes \begin{bmatrix} x_1(k-\mu) \\ x_2(k-\mu) \\ \vdots \\ x_{n-1}(k-\mu) \\ x_n(k-\mu) \end{bmatrix} \tag{3-37}$$

In short notation, the system for all μ reads

$$x(k) \geq \bigoplus_{\pm\mu=1}^{\mu_{\max}} A^{\text{syn}}_\mu(b_\mu(k - \mu)) \otimes x(k - \mu) \tag{3-38}$$

Each synchronization mode $\ell \in L_{\text{syn}}$ is coupled to unique resources i and j and provides a unique system matrix $A^{\text{syn}}_{\mu,\ell}(k - \mu)$ with only one finite element, given by

$$[A^{\text{syn}}_{\mu,\ell}(k - \mu)]_{ij} = \begin{cases} \tau_j & \text{if synchronization } \ell \text{ is made where job } k \text{ on resource } i \text{ has} \\ & \text{to be scheduled after job } k - \mu \text{ has finished on resource } j \\ \varepsilon & \text{otherwise} \end{cases} \tag{3-39}$$

The synchronization system matrix for job k with respect to $k-\mu$ can be found in Equation 3-40, which is a unique matrix for every possible combination of synchronizations between two jobs.

$$A^{\text{syn}}_\mu(b_\mu(k - \mu)) = \bigoplus_{\sigma=1}^{S} [b_\mu(k - \mu)]_\sigma \otimes A^{\text{syn}}_{\mu,\ell_\sigma}(k - \mu) \tag{3-40}$$

Chapter 4

Modeling the sorting system

The previous chapter provided a general framework to construct a model for a switching max-plus linear (SMPL) system. This chapter describes how to use the tools from that framework to make a model for Autonomous Sorting. The first steps are to define the types of choices which form the building blocks of the SMPL system and how to find them in any graph representing a sorting area. Thereafter, the constraints for routing, ordering and synchronization for the sorting system are derived. Some information on the implementation is confidential, but some code is included in the appendix. The entire model is made in Python.

4-1 Sorting areas and graphs

Prime Vision designs floor plans for each sorting area. The graph representing a random example of a floor plan is shown in Figure 4-1, on which the model is based. The model is made to fit each strongly connected graph that consists of nodes and edges. In a sorting area, there are one or more input nodes, which are the green nodes indicated by arrows in the figure, and multiple regular nodes to drive through, some of which are target nodes representing the sorting directions. The edges between nodes indicate the connections in the allowed driving direction and the edge weights represent the distances between connected nodes. In some situations, nodes have multiple incoming edges and an order needs to be chosen between two robots such that they do not enter the node at the same time. Some nodes overlap or are close to each other. In such situations, a safe distance between robots on those nodes cannot be guaranteed. Therefore, a synchronization needs to be defined, as will be discussed in section 4-2.

The model automatically defines all nodes, edges and locations for orders and synchronizations based on a given sorting area, and finds all paths starting and ending in input nodes. It divides the paths into parts in which robot orders stay the same, called segments. All these elements are the building blocks of a graph. The construction of the elements in the model is discussed in more detail in the next sections.

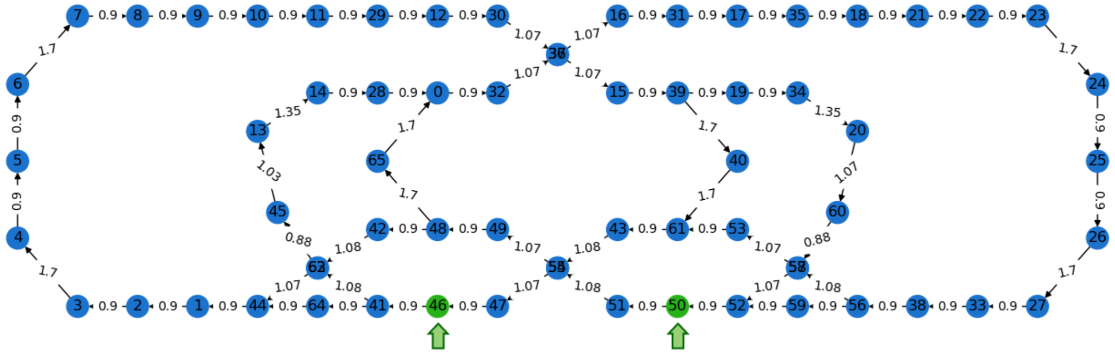


Figure 4-1: Graph on which the model is based.

4-1-1 Nodes and edges

Each node in the graph consists of a unique node ID, a node type, a position and connections to other nodes. Node types are input nodes, target nodes and regular nodes without a particular function. Connections are defined for all preceding and succeeding nodes. If there are multiple preceding nodes, the node needs to define orders between incoming robots. Nodes also contain information on other nodes that are too close to keep a safe distance between robots.

The connections between nodes are called edges, which form an important element of a weighted, directed graph. Each edge has its own weight and direction. The model constructs edges from a node and a connection to a successor node. The most important attributes of an edge are the tail and head nodes, which are the starting and ending nodes respectively, and its weight. The weight of an edge is calculated by by the model using the positions of the nodes, and represents the distance between the nodes.

4-1-2 Paths and Segments

A series of nodes and edges form a path. In the sorting system, each robot has the job to deliver a parcel. When the job is done, it returns to an input to receive a new parcel and to start a new job. Therefore, only paths starting and ending in input nodes are desired. This requires the graph to be strongly connected, such that each node can be reached from any input node, and the other way around. To model a path of a robot correctly, it should end in a node that precedes an input node. As a result, the next job starts in the input node that directly follows the last node of the previous job. This way, each node in the graph is used at most once per job. A path consists of nodes and edges, and a weight equal to the sum of all edge weights along the way.

In addition, the segments in the graph are computed. Segments are simple subsets of a path, which means that the nodes on the segment do not have multiple outgoing or incoming edges. They are found by the parts of a path between input nodes and nodes with multiple incoming or outgoing edges. If multiple robots travel the same segment, they will enter each node on the segment in the same order that they entered the first node on the segment.

4-1-3 Graphs and subgraphs

For a given sorting area, the model automatically defines the nodes, edges, paths and segments and all their useful information. Many helpful characteristics of the graph are also defined, such as lists containing all nodes of type input, and all nodes with multiple incoming and outgoing edges. Other useful characteristics are for example which segments form a path and which segments precede or succeed other segments.

Finally, the model allows to take any subgraph resembling one or more paths from the original graph. Any subgraph has the exact same structure as the original graph, but only contains the nodes that are on the desired paths. All information is maintained, and therefore any subgraph is a subset of the original graph.

4-2 Choices

Once all information for the graph is obtained, the essential elements that form the basis of the SMPL system need to be defined. This section gives an overview of all the choices that need to be made in Autonomous Sorting and how these choices are included in the system matrices of the SMPL system.

4-2-1 Choice types

The precedence graph of the total sorting area can be seen as the union of subgraphs, where each subgraph is a path. The union of two subgraphs is shown in Figure 4-2. Each robot k has to choose one subgraph to travel, which is the choice made in routing. When a robot reaches node 41, it needs to choose one of the two routes, indicated by the light blue arrows.

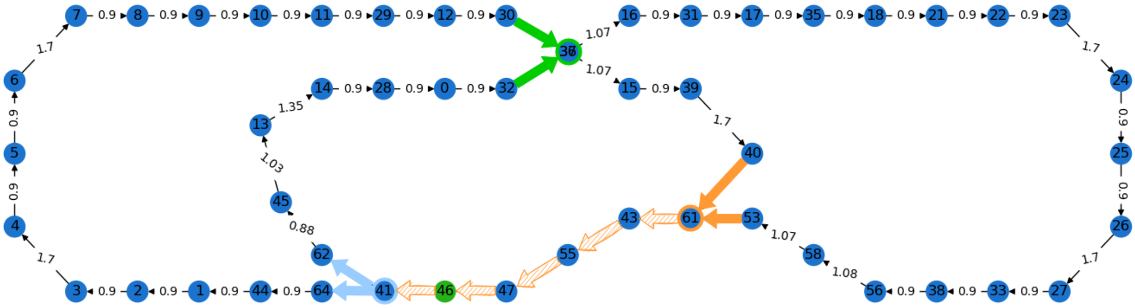


Figure 4-2: Subgraph with two paths. Different choice types are indicated with coloured arrows.

When taking the intersection of the two subgraphs, nodes and edges at common locations are left where choices need to be made. Note that nodes at the same locations are also included in the intersection, and not just nodes with the same ID, since in some situations nodes are placed on the same location with a different direction. An example is indicated with green arrows at the top middle of the graph. At this location, both subgraphs cross each other. The green arrows both point to a different node, but since they are on the same location, a synchronization choice needs to be made such that robots do not enter the nodes on the same location at the same time.

The intersection of the two graphs also results in the orange node, node 61. At this node, the two paths merge and a choice in order between robots coming from different directions needs to be made. The order between those robots is fixed for the nodes in the intersection that follow node 61, which are indicated by the empty orange arrows.

In summary, when looking at the intersection of multiple subgraphs, there are three types of choices to be made. Where the intersection has multiple connected nodes, the last node splits into the different subgraphs. This is where robots choose a route. The first node of the series of connected nodes is a node where subgraphs merge and where the order between robots from different routes needs to be chosen, which is then fixed for all nodes on that part of the intersection. Orders between robots from the same subgraph are maintained on this part as well. Where an element of the intersection consists of one node, which actually consists of two nodes on the same location, subgraphs cross and robots on the different subgraphs need to be synchronized, such that they do not cross at the same time.

4-2-2 Creating the system matrices

All choice types in an SMPL system are represented by max-plus binary variables, which are included in the system matrices. The general system matrices for routing, ordering and synchronization as given in section 3-4 are adapted to fit the sorting system. A big difference between the general structure and the SMPL system describing Autonomous Sorting is that the processing times $\tau_i(k)$ on resource i for robot k are replaced by travel times $\tau_{i,j}(k)$ from node i to node j for robot k.

Routing

The routing matrix $A^{\text{route}}(k)$ contains the weights of all edges in the graph. The weight of an edge is defined as the distance between two nodes connected by the edge, divided by the maximum velocity of the robots, representing the nominal travel time on the edge. The structure of the total routing matrix is shown in Figure 4-3, where the node numbers from 0 to 65 are on the axes. The element in row i and column j is grey if there exists an edge from node j to node i and black if not. Note that for implementation, the system matrices do not contain value ε, but the conventional algebraic equivalent 0.

Because each path has its own max-plus linear (MPL) system description, the routing matrix $A_\ell^{\text{route}}(k)$ for a specific path ℓ can be found from the complete routing matrix by keeping only the travel times for the edges that are on path ℓ. Choosing which elements are enabled and disabled is regulated by max-plus binary variables $s_{i,j}(k)$, which have value 0 if robot k travels from node i to node j, and value ε if not. Note that the matrices are defined in the

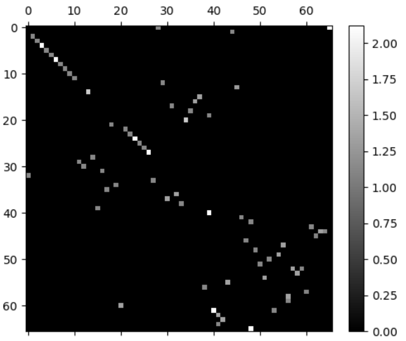
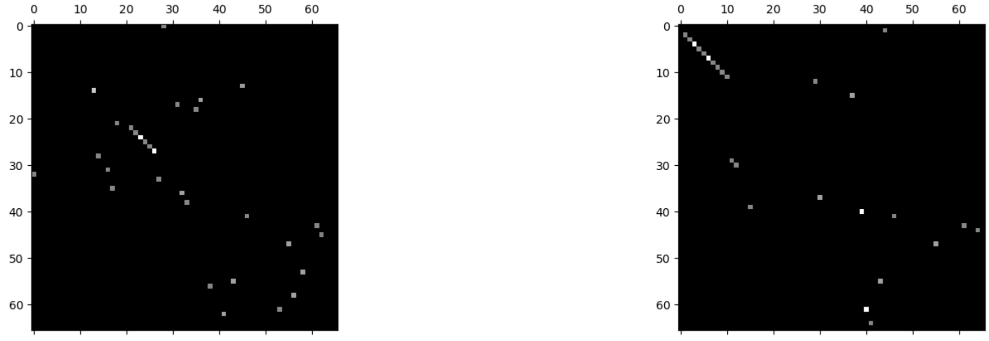


Figure 4-3: Structure of the routing matrix of the complete graph.

conventional algebra, so whenever an edge is on a path, it has a non-zero value and when it is not on a path, it becomes zero. The routing matrices for each path are also created automatically. The structures of the routing matrices of the two paths in the subgraph from Figure 4-2 are given in Figure 4-4, where path $\ell = 0$ is the path that travels the rightmost part of the graph, and $\ell = 1$ is the path that goes to the far left. Note that the set of active elements in the routing matrix of a specific path is a subset of the active elements of the complete routing matrix, depending on the edge variables $s_{i,j}(k)$.



(a) Structure of $A_0^{\text{route}}(k)$.

(b) Structure of $A_1^{\text{route}}(k)$.

Figure 4-4: Structure of routing matrices for the two paths in the subgraph.

Ordering

By replacing the processing times with travel times, $x_i(k) \otimes \tau_i(k)$ becomes $x_i(k) \otimes \tau_{i,j}(k)$ for a successor node j of node i. This last expression can be simplified to $x_j(k)$. This changes the general structure of the equations for ordering. Remember the original ordering constraints:

$$x_i(k) \geq x_i(k - \mu) \otimes \tau_i(k - \mu) \otimes z_{i,\mu}(k - \mu) \tag{4-1}$$

This can now be rewritten as

$$x_i(k) \geq x_j(k-\mu) \otimes z_{i,\mu}(k-\mu) \tag{4-2}$$

which is easier to implement as it contains less variables. This definition of order also maintains a safe distance between the robots, as they can enter a node only when the robot directly in front of them has reached a successor node. Due to this structural change in the ordering equations, the general structure of the ordering matrix as defined in Equation 3-31 also changes. Instead of a diagonal matrix, $A_\mu^{\mathrm{ord}}(z_\mu(k-\mu))$ becomes a matrix with non-zero (former non-ε) elements outside of the diagonal only. In fact, the new ordering equations relate node i to successor node j, where the routing constraints relate node i to predecessor node j. Therefore, the structure of the ordering matrix is the transpose of the complete routing matrix. Since $x_i(k) \otimes \tau_i(k)$ is replaced by $x_j(k)$, the elements of the ordering matrix are not travel times, but binary values, regulated by decision variables.

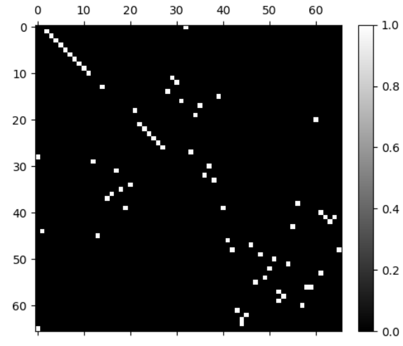


Figure 4-5: Structure of the ordering matrix.

Rows and columns in the ordering matrix with one non-zero element are nodes with one incoming and outgoing edge. On these nodes, the order between two robots is fixed and decided by the order in which the robots entered the segment that they are on, since robots cannot overtake each other. The order in these nodes is defined by the following variable $f_{l,\mu}(k-\mu)$. Rows in the ordering matrix with more than one non-zero element represent nodes with multiple outgoing edges. At these nodes, a type of ordering equations are needed to make sure that robots still keep a safe distance, even if they choose different routes. This type of ordering is called splitting. Finally, if a column in the ordering matrix has multiple non-zero elements, it concerns a node with multiple incoming edges. Here, ordering variables $z_{i,\mu}(k-\mu)$ are needed. The structure of the ordering matrix is equal for each combination of robots k and $k-\mu$, where the values of the variables with respect to those robots construct the different ordering matrices. In subsection 4-3-2, the different types of ordering variables and all ordering equations are explained in detail.

Synchronization

The same substitution of travel times instead of processing times can be made for the general synchronization equations:

$$x_d(k) \geq x_r(k-\mu) \otimes \tau_r(k-\mu) \otimes b_{i,\mu}(k-\mu) \tag{4-3}$$

becomes

$$x_d(k) \geq x_j(k-\mu) \otimes b_{i,\mu}(k-\mu) \tag{4-4}$$

where d is the successor node of i that robot k travels to, r is a node too close to d, and j is a successor node of node r. Remember the situation in Figure 4-2 where two nodes are on the same location. This is a typical situation where a synchronization occurs. In a synchronization, a robot is on a node (i) and wants to travel to a successor node (d) that has a node too close-by (r). The robot can only enter that successor node if the node close-by has been left by another robot. This is the case if the other robot that occupied the nearby node has reached its successor node (j).

Each node that has a node too close-by requires a synchronization mode, which is represented by the synchronization variable $b_{i,\mu}(k-\mu)$. For each node with such a variable, a synchronization matrix is implemented consisting of all zeros (former ε), and a synchronization variable in row d and column j. Synchronization matrices $A_{\mu}^{\text{syn}}(b_{\mu}(k-\mu))$ are automatically created for a particular node for each pair of robots k and $k-\mu$.

4-2-3 Targets and inputs

The choices that need to be made depend on an external, uncontrollable factor. Each robot is assigned a target node, which restricts the amount of possible paths. Another external factor is the input time for robot k, denoted by $u_e(k)$. To get an accurate representation of the sorting system, the chosen target nodes and input times should be based on real data. For the floor plan considered in Figure 4-1, there is no data available since the floor plan is not being used. Therefore, the assumption is made that all target nodes have equal chance to get assigned to a robot. For the input times, it is important to know what the average time is for an operator to place a parcel on a robot. A distribution for the time between two parcels is derived from data from a different floor plan. This is assumed to be representative for all operators and floor plans.

The sample data consists of approximately 1,800 parcels being sorted in a floor plan containing one input node. A histogram of the difference in time between two robots receiving a parcel $(u_e(k+1) - u_e(k))$ is shown in Figure 4-6. A gamma probability density function is fitted to the histogram, taking into account the location parameter. The location parameter defines a starting point, which is needed to give a lower boundary on the time between two parcels. An operator can not scan parcels in less time than the value of the lower boundary. The smallest value in the data set is chosen as the location parameter. The shape parameter and the scale parameter of the gamma distribution are fitted to the histogram. The input times $u_e(k)$ are drawn from the fitted distribution and applied to each input node to obtain an accurate representation of the working system. For confidentiality reasons, the numbers in the figure are changed and no information on the parameters for the fitted distribution can be given.

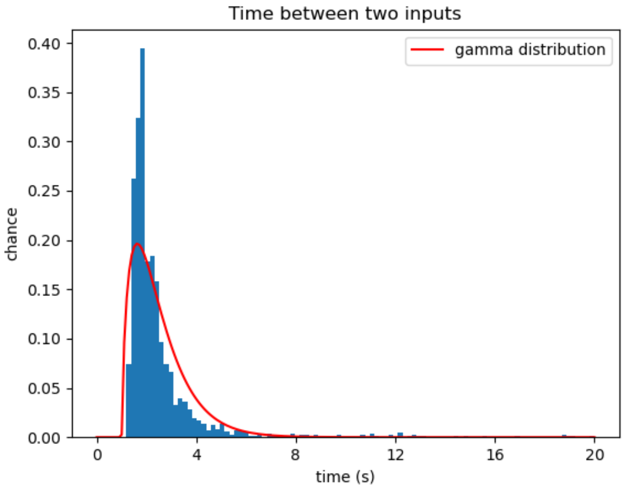


Figure 4-6: A gamma distribution fitted to a histogram of the time between two parcels.

4-3 Constraints

The general equations for all three choice types have been shown in the previous sections. For a robot entering node i, multiple constraints may be present regarding routing, ordering and synchronization. Each of these constraints is a function of max-plus linear expressions that depend on states and decision variables for all $\pm\mu = 1, \dots, \mu_{\max}$:

$$\begin{aligned} x_i(k) &\geq f^{\text{route}}(x(k), s_{i,j}(k)) \\ x_i(k) &\geq f^{\text{ord}}(x(k-\mu), z_\mu(k-\mu)) \\ x_i(k) &\geq f^{\text{syn}}(x(k-\mu), b_\mu(k-\mu)) \end{aligned} \tag{4-5}$$

The equations are inequalities, because they have to meet all the constraints and choices in order and synchronization can delay the times that robots are allowed to enter nodes. Finally, the optimal schedule will have minimal values for the states, which are equal to the maximum of all constraints [5]:

$$x_i(k) = \max(f^{\text{route}}, f^{\text{ord}}, f^{\text{syn}}) \tag{4-6}$$

How the routing, ordering and synchronization constraints are defined and how they are built from the system matrices in subsection 4-2-2 is discussed in this section. Furthermore, additional constraints that are needed to restrict the binary decision variables are defined.

From now on, successor nodes of node i are denoted by $\sigma(i)$ and predecessor nodes by $\pi(i)$.

4-3-1 Routing

The general routing constraints for robot k are defined for each node in the graph. The robot can enter node i after it has entered a preceding node. These two events must be apart at least the travel time in seconds, given that the robot is actually traveling from that predecessor to node i. The max-plus linear expressions read

$$x_i(k) \geq \bigoplus_{j \in \pi(i)} x_j(k) \otimes \tau_{j,i}(k) \otimes s_{j,i}(k) \tag{4-7}$$

This is equivalent to the conventional expressions

$$x_i(k) \geq \max_{j \in \pi(i)} x_j(k) + \tau_{j,i}(k) + \beta s^{\flat}_{j,i}(k) \tag{4-8}$$

where the max-plus operators $\oplus$ and $\otimes$ are replaced by the regular max and + operators respectively, and the max-plus binary variable $s_{j,i}(k)$ is converted to the conventional binary variable $s^{\flat}_{j,i}(k)$ with β a very large negative number. For robot k, each node in the graph gets a routing constraint for each preceding node.

Remember routing matrix $A^{\text{route}}(k)$, containing all travel times between each pair of connected nodes. All routing constraints can be created from the routing matrix, for which the implementation algorithm is shown in Algorithm 1. The implementation can be found in section A-1.

Algorithm 1 Creating routing constraints for robot k

```
for row $\in A^{\text{route}}(k)$ do                              ▷ row represents node $i$
    for $j \neq 0 \in$ row do                                    ▷ $j \in \pi(i)$
        add routing constraint for node $i$ with respect to node $j$
    end for
end for
```

The constraints are implemented in a certain structure, which is compatible with the optimization solver. More information on this can be found in subsection 5-1-2. The basic idea is that the constraints are converted to inequalities with a less than or equal to symbol, and that all known variables are placed on the right-hand side and all variables that need to be optimized on the left-hand side. Variables that need to be optimized are the states and binary decision variables. This results in the following set of routing constraints for each node i in the graph, where p is the number of predecessors of node i:

$$\begin{gathered} x_{j_1}(k) - x_i(k) + \beta s^{\flat}_{j_1,i}(k) \leq -\tau_{j_1,i}(k) \\ \vdots \\ x_{j_p}(k) - x_i(k) + \beta s^{\flat}_{j_p,i}(k) \leq -\tau_{j_p,i}(k) \end{gathered} \tag{4-9}$$

Assume that the travel times for robot k are known and therefore on the right-hand side of the constraints, even though the edge has not been traveled by the robot yet. They are given by the nominal travel times in the routing matrix. Note that at most one of the routing constraints per node i is active, since at most one edge variable can be 0 in the max-plus algebra. Due to the conversion definition to conventional algebra, given by $\beta s^{\flat}_{i,j}(k) = s_{i,j}(k)$, an edge variable is active if also the conventional variable equals 0. Therefore, $\sum_{j\in\pi(i)} s^{\flat}_{j,i}(k) = p-1$. In other words, a robot can only travel from one predecessor to i.

A special case of routing constraints is found around input nodes, where parcels enter the sorting system. The states in nodes that succeed an input node do not only depend on the time that a robot entered the input node, but also on the moment that the robot receives a parcel, denoted by $u_e(k)$. A robot at an input node has to wait there until it has received a parcel to start its job. For an input node q, this raises the following constraint:

$$x_{\sigma(q)}(k) \geq \Big(x_q(k) \oplus u_e(k)\Big) \otimes \tau_{q,\sigma(q)}(k) \otimes s_{q,\sigma(q)}(k) \tag{4-10}$$

The conventional constraints become

$$\begin{aligned} x_{\sigma(q)}(k) &\geq x_q(k) + \tau_{q,\sigma(q)}(k) + \beta s^{\flat}_{q,\sigma(q)}(k) \\ x_{\sigma(q)}(k) &\geq u_e(k) + \tau_{q,\sigma(q)}(k) + \beta s^{\flat}_{q,\sigma(q)}(k) \end{aligned} \tag{4-11}$$

Note that the top constraint is a regular routing constraint, and the bottom constraint is the input constraint.

Each robot visits a set of nodes, which leaves another set of nodes unvisited. If a certain node i stays unvisited ($x_i = 0$) for more than $\mu_{\max}$ robots in a row, the next robot k that enters node i does not take previous robots $k-\mu$ with $\mu > \mu_{\max}$ that visited node i into account. If one of those robots has not left node i yet, it will not be seen by robot k. Therefore, unvisited nodes do not get state value 0, but the latest time instant that it was visited. For each node i, the following constraint is added, where $L_i = \{\ell | i \in w_\ell\}$ is the set of paths that include node i:

$$x_i(k) \geq x_i(k-1) \otimes \bigotimes_{\ell\in L_i} \bar{w}_\ell(k) \tag{4-12}$$

If robot k travels path l that includes node i, this constraint is disabled, since $w_l(k) = 0$ and consequently, $\bar{w}_l(k) = \varepsilon$. When robot k does not visit node i, the state takes the value of the state in node i of the previous robot. In conventional algebra, the constraint becomes

$$x_i(k) \geq x_i(k-1) + \sum_{\ell\in L_i} \beta(1 - w^{\flat}_\ell(k)) \tag{4-13}$$

4-3-2 Ordering

Each node in the graph needs an ordering constraint. There are three different types of ordering constraints, as discussed in subsection 4-2-2. Nodes with a single incoming and outgoing edge need so-called following constraints, where the order between robots on a segment is fixed. Nodes with multiple outgoing edges need splitting constraints to maintain a safe distance between robots that choose different routes at that node, and nodes with multiple incoming edges need ordering constraints, where the order of robots is chosen. Each type of ordering constraints is explained in this section, starting with the following constraints.

Simple nodes

Following constraints apply to path segments that are straightforward: robots cannot enter or leave a segment other than at the start of end of the segment. For example, look at the subgraph in Figure 4-7. Generally speaking, path segments are defined as the shortest paths between input nodes, nodes with multiple incoming edges and nodes with multiple outgoing edges. The segments in this subgraph are indicated by the alternating solid and dotted lines on the orange path $\ell = 0$ and the green path $\ell = 1$. Segments are automatically found by traveling a path, starting at the input node until a node with multiple incoming or outgoing edges or an input node is reached. Orders between robots on a segment cannot be changed.

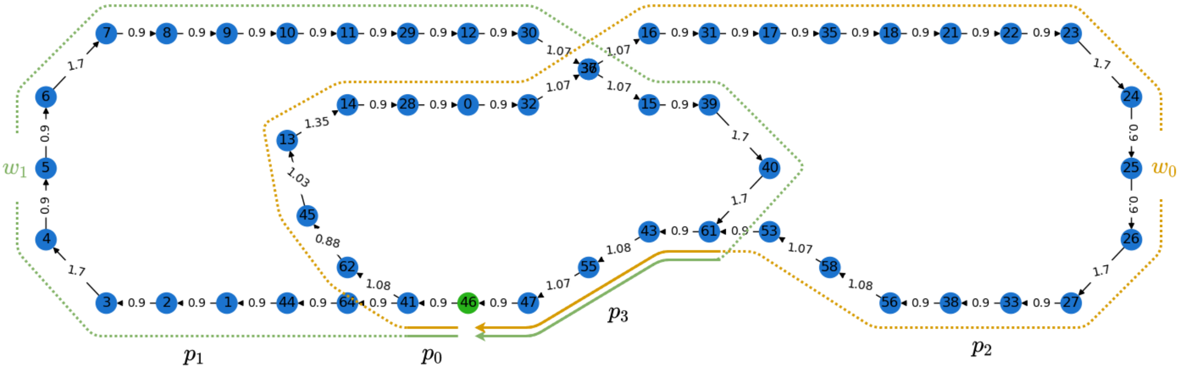


Figure 4-7: Subgraph with two paths, divided into segments.

For the following constraints, two types of max-plus binary variables are needed. Let $p_l(k)$ be the variable that denotes whether robot k uses segment $l \in \{1, \ldots, L_s\}$, where L_s is the total amount of segments in the graph. In short:

$$p_l(k) = \begin{cases} 0 & \text{if robot } k \text{ travels segment } l \\ \varepsilon & \text{if robot } k \text{ does not travel segment } l \end{cases} \tag{4-14}$$

In addition, the following variable $f_{l,\mu}(k-\mu)$ is needed which entails information on the order of robots k and $k-\mu$ on segment l:

$$f_{l,\mu}(k-\mu) = \begin{cases} 0 & \text{if robot } k-\mu \text{ enters segment } l \text{ before robot } k \\ \varepsilon & \text{otherwise} \end{cases} \tag{4-15}$$

Following constraints are defined on all nodes on a segment but the last, since the last nodes are also the first node of the next segment(s), for which following constraints will be defined. In following constraints, the moment that robot k can enter a node on a segment depends on the time that other robots on the same segment in front of robot k have reached a successor node. Therefore, following constraints are active when both robots travel the same segment. The following constraints are given in Equation 4-16.

$$\begin{aligned} x_i(k) &\geq x_{\sigma(i)}(k-\mu) \otimes f_{l,\mu}(k-\mu) \otimes p_l(k) \otimes p_l(k-\mu) \\ x_i(k-\mu) &\geq x_{\sigma(i)}(k) \otimes \bar{f}_{l,\mu}(k-\mu) \otimes p_l(k) \otimes p_l(k-\mu) \end{aligned} \tag{4-16}$$

The constraints are defined for all nodes i on segment l but the last and all $\pm\mu = 1, \dots, \mu_{\max}$. Each i has only one successor node since the segment is a path where each node has only one outgoing edge, until possibly the last.

The following constraints for robot k, taking into account all robots in the range $\mu_{\max}$ that started their job later and earlier than robot k, are summarized as follows, for each node i and for all segments $l = \{1, \dots, L_s\}$:

$$\begin{aligned} x_i(k) \geq \bigoplus_{\mu=1}^{\mu_{\max}} \Big(& x_{\sigma(i)}(k-\mu) \otimes f_{l,\mu}(k-\mu) \otimes p_l(k) \otimes p_l(k-\mu) \oplus \\ & x_{\sigma(i)}(k+\mu) \otimes \bar{f}_{l,\mu}(k) \otimes p_l(k+\mu) \otimes p_l(k) \Big) \end{aligned} \tag{4-17}$$

The union of all segments contains all nodes in the graph, so each node is sure to have a following constraint. Multiple following constraints occur for nodes with multiple outgoing edges, since those types of nodes are the starting point of multiple segments. This happens for example in node 41 in the subgraph, where both p_1 and p_2 start.

To convert the max-plus linear expressions for following into conventional constraints, the max-plus binary following and segment variables are replaced by conventional binary variables in the same way as in Equation 3-3. For example, the following variables are linked as

$$f_{l,\mu}(\cdot) = \begin{cases} 0 & \text{if } f^{\flat}_{l,\mu}(\cdot) = 0 \\ \varepsilon & \text{if } f^{\flat}_{l,\mu}(\cdot) = 1 \end{cases} \qquad \bar{f}_{l,\mu}(\cdot) = \begin{cases} \varepsilon & \text{if } f^{\flat}_{l,\mu}(\cdot) = 0 \\ 0 & \text{if } f^{\flat}_{l,\mu}(\cdot) = 1 \end{cases} \tag{4-18}$$

The conventional following constraints are formulated as

$$\begin{aligned} x_i(k) \geq \max_{\mu=1,\dots,\mu_{\max}} \Big(& x_{\sigma(i)}(k-\mu) + \beta f^{\flat}_{l,\mu}(k-\mu) + \beta p^{\flat}_l(k) + \beta p^{\flat}_l(k-\mu) \;, \\ & x_{\sigma(i)}(k+\mu) + \beta(1 - f^{\flat}_{l,\mu}(k)) + \beta p^{\flat}_l(k+\mu) + \beta p^{\flat}_l(k) \Big) \end{aligned} \tag{4-19}$$

Equation 4-19 consists of many single inequalities for each node i on segment l:

$$\begin{aligned} x_i(k) &\geq x_{\sigma(i)}(k-1) + \beta f^{\flat}_{l,1}(k-1) + \beta p^{\flat}_l(k) + \beta p^{\flat}_l(k-1) \\ x_i(k) &\geq x_{\sigma(i)}(k+1) + \beta(1 - f^{\flat}_{l,1}(k)) + \beta p^{\flat}_l(k+1) + \beta p^{\flat}_l(k) \\ &\;\vdots \\ x_i(k) &\geq x_{\sigma(i)}(k-\mu_{\max}) + \beta f^{\flat}_{l,\mu_{\max}}(k-\mu_{\max}) + \beta p^{\flat}_l(k) + \beta p^{\flat}_l(k-\mu_{\max}) \\ x_i(k) &\geq x_{\sigma(i)}(k+\mu_{\max}) + \beta(1 - f^{\flat}_{l,\mu_{\max}}(k)) + \beta p^{\flat}_l(k+\mu_{\max}) + \beta p^{\flat}_l(k) \end{aligned} \tag{4-20}$$

Each node i gets an amount of $2 \cdot \mu_{\max}$ following constraints for each segment that it belongs to, except for the last node in the segment, since it gets its following constraints from the next segment.

To illustrate the importance following constraints, consider two robots. First, assume that robot k and $k-1$ travel the same path, say w_1, as indicated by the green path in Figure 4-7. This path consists of segments 0, 1 and 3. Since both k and $k-1$ travel on w_1, the values of the segment variables are as shown in Table 4-1.

	Segment			
Robot	0	1	2	3
$k-1$	0	0	ε	0
k	0	0	ε	0

Table 4-1: Segment variables for robots $k-1$ and k travelling the same path.

The following constraints for robot k and $k-1$ are shown in Equation 4-21 for all nodes i_l and their successor j_l on segment l.

$$
\begin{aligned}
x_{i_0}(k) &\geq x_{j_0}(k-1) \otimes f_{0,1}(k-1) \otimes p_0(k) \otimes p_0(k-1) \\
x_{i_0}(k-1) &\geq x_{j_0}(k) \otimes \bar{f}_{0,1}(k-1) \otimes p_0(k) \otimes p_0(k-1) \\
\\
x_{i_1}(k) &\geq x_{j_1}(k-1) \otimes f_{1,1}(k-1) \otimes p_1(k) \otimes p_1(k-1) \\
x_{i_1}(k-1) &\geq x_{j_1}(k) \otimes \bar{f}_{1,1}(k-1) \otimes p_1(k) \otimes p_1(k-1) \\
\\
x_{i_2}(k) &\geq x_{j_2}(k-1) \otimes f_{2,1}(k-1) \otimes p_2(k) \otimes p_2(k-1) \\
x_{i_2}(k-1) &\geq x_{j_2}(k) \otimes \bar{f}_{2,1}(k-1) \otimes p_2(k) \otimes p_2(k-1) \\
\\
x_{i_3}(k) &\geq x_{j_3}(k-1) \otimes f_{3,1}(k-1) \otimes p_3(k) \otimes p_3(k-1) \\
x_{i_3}(k-1) &\geq x_{j_3}(k) \otimes \bar{f}_{3,1}(k-1) \otimes p_3(k) \otimes p_3(k-1)
\end{aligned}
\tag{4-21}
$$

By definition, robot $k-1$ started its job before k, so the following variable $f_{l,1}(k-1) = 0$ for each l on the path, and thus $\bar{f}_{l,1}(k-1) = \varepsilon$ for segments $l = 0, 1, 3$. Only for segment 2, $f_{2,1}(k-1) = \varepsilon$ and $\bar{f}_{2,1}(k-1) = 0$. Still, the constraints for nodes on segment 2 are inactive, due to the segment variables $p_2(k)$ and $p_2(k-1)$. This can be seen by substituting the variable values into all constraints:

$$
\begin{aligned}
x_{i_0}(k) \geq x_{j_0}(k-1) \otimes 0 \otimes 0 \otimes 0 \quad &\Longrightarrow \quad x_{i_0}(k) \geq x_{j_0}(k-1) \\
x_{i_0}(k-1) \geq x_{j_0}(k) \otimes \varepsilon \otimes 0 \otimes 0 \quad &\Longrightarrow \quad x_{i_0}(k-1) \geq \varepsilon \\
\\
x_{i_1}(k) \geq x_{j_1}(k-1) \otimes 0 \otimes 0 \otimes 0 \quad &\Longrightarrow \quad x_{i_1}(k) \geq x_{j_1}(k-1) \\
x_{i_1}(k-1) \geq x_{j_1}(k) \otimes \varepsilon \otimes 0 \otimes 0 \quad &\Longrightarrow \quad x_{i_1}(k-1) \geq \varepsilon \\
\\
x_{i_2}(k) \geq x_{j_2}(k-1) \otimes \varepsilon \otimes \varepsilon \otimes \varepsilon \quad &\Longrightarrow \quad x_{i_2}(k) \geq \varepsilon \\
x_{i_2}(k-1) \geq x_{j_2}(k) \otimes 0 \otimes \varepsilon \otimes \varepsilon \quad &\Longrightarrow \quad x_{i_2}(k-1) \geq \varepsilon \\
\\
x_{i_3}(k) \geq x_{j_3}(k-1) \otimes 0 \otimes 0 \otimes 0 \quad &\Longrightarrow \quad x_{i_3}(k) \geq x_{j_3}(k-1) \\
x_{i_3}(k-1) \geq x_{j_3}(k) \otimes \varepsilon \otimes 0 \otimes 0 \quad &\Longrightarrow \quad x_{i_3}(k-1) \geq \varepsilon
\end{aligned}
\tag{4-22}
$$

As expected, only the constraints on segments 0, 1 and 3 are active for robot k, since it is driving behind robot $k-1$ on the same path. Constraints on segment 2 are neglected, since neither of the robots travel on that segment.

Now assume that both robots travel a different path. Say robot $k-1$ travels on path 1, and robot k on path 0. The values of the segment variables in this case are shown in Table 4-2.

	Segment			
Robot	0	1	2	3
$k-1$	0	0	ε	0
k	0	ε	0	0

Table 4-2: Segment variables for robots $k-1$ travelling path $\ell = 1$ and k path $\ell = 0$.

Substituting the variables into the following constraints results in

$$\begin{aligned}
x_{i_0}(k) &\geq x_{j_0}(k-1) \otimes 0 \otimes 0 \otimes 0 &&\Longrightarrow && x_{i_0}(k) \geq x_{j_0}(k-1)\\
x_{i_0}(k-1) &\geq x_{j_0}(k) \otimes \varepsilon \otimes 0 \otimes 0 &&\Longrightarrow && x_{i_0}(k-1) \geq \varepsilon\\[1em]
x_{i_1}(k) &\geq x_{j_1}(k-1) \otimes \varepsilon \otimes \varepsilon \otimes 0 &&\Longrightarrow && x_{i_1}(k) \geq \varepsilon\\
x_{i_1}(k-1) &\geq x_{j_1}(k) \otimes 0 \otimes \varepsilon \otimes 0 &&\Longrightarrow && x_{i_1}(k-1) \geq \varepsilon\\[1em]
x_{i_2}(k) &\geq x_{j_2}(k-1) \otimes \varepsilon \otimes 0 \otimes \varepsilon &&\Longrightarrow && x_{i_2}(k) \geq \varepsilon\\
x_{i_2}(k-1) &\geq x_{j_2}(k) \otimes 0 \otimes 0 \otimes \varepsilon &&\Longrightarrow && x_{i_2}(k-1) \geq \varepsilon
\end{aligned} \tag{4-23}$$

$$\begin{aligned}
x_{i_3}(k) &\geq x_{j_3}(k-1) \otimes f_{3,1}(k-1) \otimes 0 \otimes 0 &&\Longrightarrow && x_{i_3}(k) \geq x_{j_3}(k-1) \otimes f_{3,1}(k-1)\\
x_{i_3}(k-1) &\geq x_{j_3}(k) \otimes \bar{f}_{3,1}(k-1) \otimes 0 \otimes 0 &&\Longrightarrow && x_{i_3}(k-1) \geq x_{j_3}(k) \otimes \bar{f}_{3,1}(k-1)
\end{aligned} \tag{4-24}$$

Note that following constraints for segments are always inactive if at most one of the robots travels the segment, since then at least one of the segment variables $p_l(k)$ or $p_l(k-\mu)$ equals ε. Furthermore, both robots drive on segment 3 after they have been apart, so a new robot order needs to be chosen on segment 3. Which one of the constraints in Equation 4-24 gets activated, is decided by the following variable $f_{3,1}(k-1)$. This situation will be studied in the paragraph on nodes with multiple incoming edges.

Nodes with multiple outgoing edges

The following constraints ensure a safe distance between robots that travel the same segment. If two robots reach the end of a common segment, but choose different successor segments, the distance between the robots in the last node of the common segment is not protected by following constraints, since the robots choose different successor nodes. Therefore, splitting constraints are needed, where the order of the robots in the previous segment is taken into account. For each successor node j of node i with multiple outgoing edges, and for all segments l that finish in node i, the basic splitting constraints are given by

$$\begin{aligned}
x_i(k) &\geq x_j(k-\mu) \otimes f_{l,\mu}(k-\mu)\\
x_i(k-\mu) &\geq x_j(k) \otimes \bar{f}_{l,\mu}(k-\mu)
\end{aligned} \tag{4-25}$$

The total set of splitting constraints for robot k, taking into account a range of $\mu_{\max}$ robots that started their job earlier and later than k, are given by

$$x_i(k) \geq \bigoplus_{j\in\sigma(i)} \bigoplus_{l\in\pi_s(i)} \bigoplus_{\mu=1}^{\mu_{\max}} \Big(x_j(k-\mu) \otimes f_{l,\mu}(k-\mu) \oplus \\ x_j(k+\mu) \otimes \bar{f}_{l,\mu}(k) \Big) \tag{4-26}$$

Here, $\sigma(i)$ is the set of all successor nodes of node i and $\pi_s(i)$ is the set of segments that end in node i. Translating this into conventional constraints gives

$$x_i(k) \geq \max_{j\in\sigma(i)} \max_{l\in\pi_s(i)} \max_{\mu=1,\ldots,\mu_{\max}} \Big(x_j(k-\mu) + \beta f^{\flat}_{l,\mu}(k-\mu)\ , \\ x_j(k+\mu) + \beta(1 - f^{\flat}_{l,\mu}(k)) \Big) \tag{4-27}$$

Since these constraints hold for each successor node j of i with multiple outgoing edges, and for each segment l that ends in node i, node i has an amount of $2 \cdot \mu_{\max} \cdot |\pi_s(i)| \cdot |\sigma(i)|$ separate max-plus linear expressions for splitting.

An example of a node that needs splitting constraints is node 41 in the subgraph, a closeup of which is shown in Figure 4-8. Node 41 is the only node in the subgraph with multiple outgoing edges, of which it has two ($|\sigma(41)| = 2$), and it has one incoming segment ($|\pi_s(41)| = 1$). Therefore, the total amount of splitting constraints for the subgraph is $4 \cdot \mu_{\max}$.

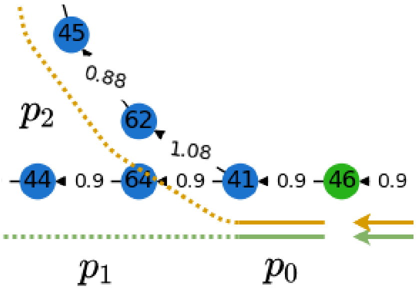


Figure 4-8: Two segments start in node 41.

Say that two robots $k-1$ and k enter node 41, and robot $k-1$ is in front of k. Therefore, $f_{0,1}(k-1) = 0$ and $\bar{f}_{0,1}(k-1) = \varepsilon$. If both robots continue to travel on the same segment after node 41, their movements are covered by the following constraints. On the other hand, if they choose different segments, the following constraints are deactivated, as seen in the example in Equation 4-23. Robot k still needs constraints to wait until robot $k-1$ has left node 41, which is done by the splitting constraints. Substituting the variables into Equation 4-25 results in

$$\begin{aligned}
x_{41}(k) &\geq x_{62}(k-1) \otimes f_{0,1}(k-1) &&\implies && x_{41}(k) \geq x_{62}(k-1)\\
x_{41}(k-1) &\geq x_{62}(k) \otimes \bar{f}_{0,1}(k-1) &&\implies && x_{41}(k-1) \geq \varepsilon\\
\\
x_{41}(k) &\geq x_{64}(k-1) \otimes f_{0,1}(k-1) &&\implies && x_{41}(k) \geq x_{64}(k-1)\\
x_{41}(k-1) &\geq x_{64}(k) \otimes \bar{f}_{0,1}(k-1) &&\implies && x_{41}(k-1) \geq \varepsilon
\end{aligned} \tag{4-28}$$

Note that only one of the remaining constraints is active, since robot $k-1$ can not enter both node 62 and 64.

Nodes with multiple incoming edges

Remember the segments defined in the subgraph in Figure 4-7, where segments 1 and 2 both continue to segment 3. The constraints for segment 3 depend on the order in which the robots enter this segment, but this is yet to be determined. This is decided by max-plus binary ordering variable $z_{i,\mu}(k-\mu)$:

$$z_{i,\mu}(k-\mu) = \begin{cases} 0 & \text{if robot } k-\mu \text{ enters node } i \text{ before robot } k \\ \varepsilon & \text{otherwise} \end{cases} \tag{4-29}$$

This control variable is used for nodes with multiple incoming edges. These nodes are located at the start of a segment and already have following constraints as defined before, but they need a new type of constraint in case two robots enter the segment from two different segments. These constraints are called ordering constraints and they are defined in Equation 4-30.

$$\begin{aligned}
x_i(k) &\geq x_j(k-\mu) \otimes z_{i,\mu}(k-\mu) \otimes p_l(k) \otimes p_m(k-\mu)\\
x_i(k-\mu) &\geq x_j(k) \otimes \bar{z}_{i,\mu}(k-\mu) \otimes p_l(k) \otimes p_m(k-\mu)
\end{aligned} \tag{4-30}$$

The ordering constraints are defined for all successor nodes j of node i with multiple incoming edges, and for all segments $l \neq m$ that finish in node i. The constraints make sure that a robot has to wait to enter node i until earlier robots, coming from another segment, have left i. The constraints hold for each combination of robots k and $k-\mu$ for $\mu = \pm 1, \ldots, \mu_{\max}$ and can be summarized for robot k:

$$\begin{aligned}
x_i(k) \geq \bigoplus_{j \in \sigma(i)} \bigoplus_{l,m \in \pi_s(i)} \bigoplus_{\mu=1}^{\mu_{\max}} \Big(& x_j(k-\mu) \otimes z_{i,\mu}(k-\mu) \otimes p_l(k) \otimes p_m(k-\mu) \oplus \\
& x_j(k+\mu) \otimes \bar{z}_{i,\mu}(k) \otimes p_l(k+\mu) \otimes p_m(k) \Big), \quad l \neq m
\end{aligned} \tag{4-31}$$

Using the same method to replace the max-plus binary ordering variables with conventional binary variables as before, the conventional ordering constraints are given by:

$$
\begin{aligned}
x_i(k) \geq \max_{j\in\sigma(i)} \max_{l,m\in\pi_s(i)} \max_{\mu=1,\ldots,\mu_{\max}} \Big(& x_j(k-\mu) + \beta z^\flat_{i,\mu}(k-\mu) + \beta p^\flat_l(k) + \beta p^\flat_m(k-\mu) \ , \\
& x_j(k+\mu) + \beta(1 - z^\flat_{i,\mu}(k)) + \beta p^\flat_l(k+\mu) + \beta p^\flat_m(k) \Big) \\
& l \neq m
\end{aligned}
\tag{4-32}
$$

These constraints hold for each successor node j of i and for each combination of different segments l and m that end in node i. Therefore, each node i with multiple incoming edges has a total amount of $2 \cdot \mu_{\max} \cdot |\sigma(i)| \cdot |\pi_s(i)| \cdot (|\pi_s(i)| - 1)$ ordering constraints.

An example of ordering is given in Figure 4-9. Node 61 has one outgoing edge ($|\sigma(61)| = 1$) and two incoming segments ($|\pi_s(61)| = 2$), so the total amount of ordering constraints for the subgraph is $4 \cdot \mu_{\max}$. Assume that robot $k-1$ travels on segment 1 and robot k on segment 2. Both segments join in node 61, and the decision which robot can enter first is made by $z_{61,1}(k-1)$.

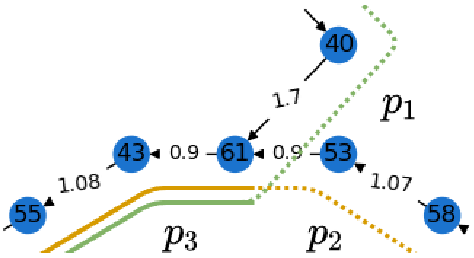


Figure 4-9: Orders of robots need to be defined in node 61.

The set of ordering constraints for node 61 becomes

$$
\begin{aligned}
x_{61}(k) &\geq x_{43}(k-1) \otimes z_{61,1}(k-1) \otimes p_1(k) \otimes p_2(k-1) \\
x_{61}(k-1) &\geq x_{43}(k) \otimes \bar{z}_{61,1}(k-1) \otimes p_1(k) \otimes p_2(k-1) \\
\\
x_{61}(k) &\geq x_{43}(k-1) \otimes z_{61,1}(k-1) \otimes p_2(k) \otimes p_1(k-1) \\
x_{61}(k-1) &\geq x_{43}(k) \otimes \bar{z}_{61,1}(k-1) \otimes p_2(k) \otimes p_1(k-1)
\end{aligned}
\tag{4-33}
$$

and substituting the known variables, this results in

$$
\begin{aligned}
x_{61}(k) &\geq x_{43}(k-1) \otimes z_{61,1}(k-1) \otimes \varepsilon \otimes \varepsilon \\
x_{61}(k-1) &\geq x_{43}(k) \otimes \bar{z}_{61,1}(k-1) \otimes \varepsilon \otimes \varepsilon \\
\\
x_{61}(k) &\geq x_{43}(k-1) \otimes z_{61,1}(k-1) \otimes 0 \otimes 0 \\
x_{61}(k-1) &\geq x_{43}(k) \otimes \bar{z}_{61,1}(k-1) \otimes 0 \otimes 0
\end{aligned}
\tag{4-34}
$$

Note that ordering constraints are only active for a combination of robots if they approach the node from different edges. Control variable $z_{61,1}(k-1)$ decides which robot enters node

61 first. Say that robot $k-1$ can go first. Then, according to Equation 4-29, $z_{61,1}(k-1) = 0$, and thus $\bar{z}_{61,1}(k-1) = \varepsilon$. The ordering constraints from Equation 4-34 become

$$\begin{aligned} x_{61}(k) &\geq x_{43}(k-1) \\ x_{61}(k-1) &\geq \varepsilon \end{aligned} \tag{4-35}$$

Remember the example of following constraints from Equation 4-24, where the following variable $f_{3,1}(k-1)$ is still unknown. For each node on p_3, the following constraints include this variable and it is actually defined by the order in which the robots enter the first node of the segment. That decision has been made by the ordering control variable $z_{61,1}(k-1)$, and therefore $f_{3,1}(k-1)$ should take the same value.

Implementation

Remember ordering matrix $A_\mu^{\text{ord}}(k)$ from subsection 4-2-2. Each non-zero element in the matrix resembles an ordering variable with respect to two nodes; the node with multiple incoming edges and a successor node. All ordering constraints can be created from the ordering matrix. The implementation algorithm is shown in Algorithm 2, and the implementation in Python can be found in section A-2.

Algorithm 2 Creating ordering constraints for robot k

```
for row ∈ $A_\mu^{\text{ord}}(k)$ do                                      ▷ row represents node $i$
    if row has > 1 non-zero elements then                          ▷ $i$ has > 1 outgoing edges
        for $j \neq 0 \in$ row do                                          ▷ $j \in \sigma(i)$
            for $\mu = 1, \dots, \mu_{\max}$ do
                add splitting constraint for robot $k$ and $k - \mu$
                add splitting constraint for robot $k$ and $k + \mu$
            end for
        end for
    else if $[A_\mu^{\text{ord}}(k)]_{:,\text{row}}$ has > 1 non-zero elements then      ▷ $i$ has > 1 incoming edges
        for $p_l \in \pi_s(\text{row})$ do
            for $p_m \in \pi_s(\text{row}) \setminus \{p_l\}$ do          ▷ each combination of preceding segments
                for $\mu = 1, \dots, \mu_{\max}$ do
                    add ordering constraint for robot $k$ and $k - \mu$
                    add ordering constraint for robot $k$ and $k + \mu$
                end for
            end for
        end for
    else                                                ▷ $i$ has 1 incoming and outgoing edge
        for $\mu = 1, \dots, \mu_{\max}$ do
            add following constraint for robot $k$ and $k - \mu$
            add following constraint for robot $k$ and $k + \mu$
        end for
    end if
end for
```

4-3-3 Synchronization

In situations where robots have to wait for each other in different nodes, synchronization constraints are needed. This is the case for configurations where nodes are too close to each other to keep a safe distance between robots. In addition, a concept called coupling is also a type of synchronization. Coupling is an action that occurs at input nodes, where a robot k starts a new job and therefore changes into a robot $k+\gamma$. Both synchronization types are discussed in this section.

Safe distance

Some nodes block the availability of another node until the blocking node is free again. This happens for nodes that are too close to each other, which therefore require synchronization constraints. An example of two paths crossing is shown in Figure 4-10, where node 36 and 37 in the middle are on the same location. For a robot on node 30 to continue to node 37, node 36 has to be left by other robots driving there. The same holds for a robot on node 32, which has to wait before continuing to node 36 until other robots have left node 37.

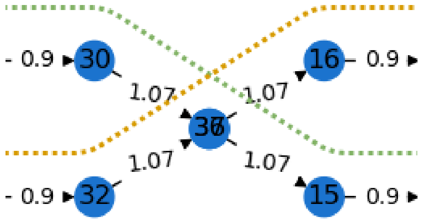


Figure 4-10: At crossings, synchronization constraints are needed.

To be able to choose which robots can go first in these situations, the max-plus binary synchronization variable $b_{i,\mu}(k-\mu)$ is defined:

$$b_{i,\mu}(k-\mu) = \begin{cases} 0, & \text{if robot } k \text{ has to wait in node } i \text{ until robot } k-\mu \text{ has} \\ & \text{left the node that is too close to a successor node of } i \\ \varepsilon, & \text{otherwise} \end{cases} \tag{4-36}$$

Synchronization variables are defined for nodes i in which robots might have to wait for other robots due to the close proximity of a node to a successor node of i. The nodes that block the availability of a successor node of i are denoted by $b(i)$. The synchronization constraints are defined as:

$$\begin{aligned} x_j(k) &\geq x_d(k-\mu) \otimes b_{i,\mu}(k-\mu) \otimes s_{i,j}(k) \otimes s_{c,d}(k-\mu) \\ x_j(k-\mu) &\geq x_d(k) \otimes \bar{b}_{i,\mu}(k-\mu) \otimes s_{c,d}(k) \otimes s_{i,j}(k-\mu) \end{aligned} \tag{4-37}$$

They hold for all successor nodes j of node i that have another node too close-by, for all nodes c that are too close to j, and for all successor nodes d of c. With these constraints, robots are sure to wait in a node until other robots have reached the node that comes after the node that is too close to the waiting robot's next node. This way, no conflicts will occur at synchronization configurations like crossings. The synchronization constraints for robot k are summarized as follows:

$$\begin{aligned} x_j(k) \geq \bigoplus_{c\in b(i)} \bigoplus_{d\in\sigma(c)} \bigoplus_{\mu=1,\dots,\mu_{\max}} \Big(& x_d(k-\mu) \otimes b_{i,\mu}(k-\mu) \otimes s_{i,j}(k) \otimes s_{c,d}(k-\mu) \oplus \\ & x_d(k+\mu) \otimes \bar{b}_{i,\mu}(k) \otimes s_{c,d}(k+\mu) \otimes s_{i,j}(k) \Big) \end{aligned} \tag{4-38}$$

They are defined for each node i that needs a synchronization, and each successor j of i for which nodes exist that are too close-by. In general, node i has a total of $2 \cdot \mu_{\max} \cdot |\sigma(i)| \cdot |b(i)| \cdot |\sigma(b(i))|$ synchronization constraints. The conventional constraints are given by

$$\begin{aligned} x_j(k) \geq \max_{c\in b(i)} \max_{d\in\sigma(c)} \max_{\mu=1,\dots,\mu_{\max}} \Big(& x_d(k-\mu) + \beta b^\flat_{i,\mu}(k-\mu) + \beta s^\flat_{i,j}(k) + \beta s^\flat_{c,d}(k-\mu) \ , \\ & x_d(k+\mu) + \beta(1 - b^\flat_{i,\mu}(k)) + \beta s^\flat_{c,d}(k+\mu) + \beta s^\flat_{i,j}(k) \Big) \end{aligned} \tag{4-39}$$

Synchronization variables are assigned to all nodes in one configuration where a robot should be able to wait, but actually only one variable is needed. To illustrate, a crossing configuration is shown in Figure 4-11 where robots k and $k+1$ both want to cross. Robot k travels from node 30 via 37 to 15 and robot $k+1$ from 32 via 36 to 16. The constraints for both robots, substituting into Equation 4-38 for $\mu_{\max} = 1$, are given in Equation 4-40.

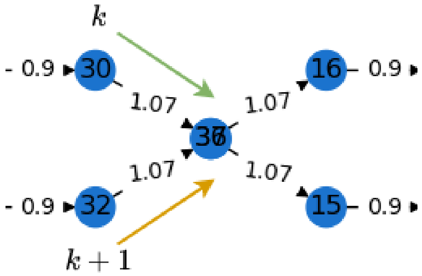


Figure 4-11: Robots k and $k+1$ use the same synchronization variable at a crossing.

$$\begin{aligned} x_{37}(k) \geq\ & x_{16}(k-1) \otimes b_{30,1}(k-1) \otimes s_{30,37}(k) \otimes s_{36,16}(k-1) \oplus \\ & x_{16}(k+1) \otimes \bar{b}_{30,1}(k) \otimes s_{36,16}(k+1) \otimes s_{30,37}(k) \\ x_{36}(k+1) \geq\ & x_{15}(k) \otimes b_{32,1}(k) \otimes s_{32,36}(k+1) \otimes s_{37,15}(k) \oplus \\ & x_{15}(k+2) \otimes \bar{b}_{32,1}(k+1) \otimes s_{37,15}(k+2) \otimes s_{32,36}(k+1) \end{aligned} \tag{4-40}$$

There are two constraints regarding robots k and $k+1$, and precisely one of these constraints should be active. Since the edge variables $(s_{36,16}(k+1) = s_{30,37}(k) = s_{32,36}(k+1) = s_{37,15}(k))$

are all active, this solely depends on the max-plus binary synchronization variables $\bar{b}_{30,1}(k)$ and $b_{32,1}(k)$, of which precisely one should be 0. Therefore, it is necessary that $b_{30,1}(k) = b_{32,1}(k)$ and this should hold for every k and all μ. For any configuration with nodes too close to each other, only one synchronization variable is needed.

Implementation

In subsection 4-2-2, it was mentioned that each node with a synchronization mode has its own synchronization matrix $A^{\text{syn}}_{\mu}(k)$ with one non-zero element. This element is one of the conventional binary synchronization variables that represents the order of robots k and $k - \mu$ in the configuration that node is a part of. The steps to create synchronization constraints from the matrices are presented in the pseudo-code in Algorithm 3. The implementation is similar to the implementation of the ordering constraints.

Algorithm 3 Creating synchronization constraints for robot k

```
for node $i$ with synchronization matrix do
    for $j \in \sigma(i)$ with nodes $b(i)$ that are too close-by do
        for close node $c \in b(i)$ do
            for $d \in \sigma(c)$ do
                for $\mu = 1, \dots, \mu_{\max}$ do
                    add synchronization constraint for robot $k$ and $k - \mu$
                    add synchronization constraint for robot $k$ and $k + \mu$
                end for
            end for
        end for
    end for
end for
```

Coupling

When robots have delivered their parcel, they finish their job by driving back to an input node. When robot k returns to a node preceding an input, its job is complete. This means that at the succeeding input node, the same robot starts a new job $k + \gamma$. The robot from job k needs to be coupled to $k + \gamma$ at input node x_q. To find the correct value for γ, new max-plus binary variables are needed.

Say that a constant amount of r robots is driving through the graph, and the active robots at a time instant are $k, k + 1, \dots, k + r - 1$. When robot k is the first to return at an input, robot $k + r$ starts its job at that input. This is indicated by the coupling variable, where $\gamma = 1, \dots, \gamma_{\max}$:

$$c_k(k+\gamma) = \begin{cases} 0 & \text{if robot } k \text{ is coupled to robot } k+\gamma \\ \varepsilon & \text{otherwise} \end{cases} \tag{4-41}$$

This would mean that $c_k(k + r) = 0$ if robot k is the first robot to finish its job. Note that no other robots can be coupled to $k + r$. In addition, all other variables that couple k to a new robot should also be disabled, since robot k can only be coupled once. The coupling variables are included in constraints on input nodes q for robot k for all predecessor nodes j of input node q and for all $\gamma = 1, \ldots, \gamma_{\max}$:

$$x_q(k) \geq x_j(k-\gamma) \otimes \tau_{j,q}(k-\gamma) \otimes s_{j,q}(k-\gamma) \otimes c_{k-\gamma}(k) \tag{4-42}$$

The constraint ensures that robot k can enter an input node only after another robot has entered a preceding node and has traveled an edge to the input, under the conditions that the other robot is coupled to robot k. This can be summarized for all inputs as

$$x_q(k) \geq \bigoplus_{j \in \pi(q)} \bigoplus_{\gamma=1}^{\gamma_{\max}} x_j(k-\gamma) \otimes \tau_{j,q}(k-\gamma) \otimes s_{j,q}(k-\gamma) \otimes c_{k-\gamma}(k) \tag{4-43}$$

In conventional algebra, this reads

$$x_q(k) \geq \max_{j \in \pi(q)} \max_{\gamma=1,\ldots,\gamma_{\max}} \Big(x_j(k-\gamma) + \tau_{j,q}(k-\gamma) + \beta s^\flat_{j,q}(k-\gamma) + \beta c^\flat_{k-\gamma}(k) \Big) \tag{4-44}$$

Earlier, following and ordering constraints were formulated for all nodes in the graph, keeping a safe distance between each pair of robots that differ at most $\mu_{\max}$ in index. Since the robot index changes at input nodes due to coupling, robots at nodes that precede inputs might not take into account a newly coupled robot that is present at an input node. The distance between two robots between an input node and preceding nodes can therefore get too small. This issue can be resolved by defining new constraints for each predecessor node p of input node q, and for each edge (p, i) and (q, j):

$$x_p(k) \geq x_q(k+\gamma) \otimes c_{k+\mu}(k+\gamma) \otimes s_{p,i}(k) \otimes s_{q,j}(k+\gamma) \tag{4-45}$$

The constraints are considered for all $\gamma = 1, \ldots, r$. If γ is larger than r, a situation can occur that robot k must wait for robot $k+\gamma$, while they are actually the same robot. In other words, there is a $0 < \kappa < \gamma$ such that $c_k(k+\kappa) = 0$ and $c_{k+\kappa}(k+\gamma) = 0$. This makes the model unsolvable. In addition, all $\mu = r-1, \ldots, r-\gamma_{\max}$, but $\mu \neq 0$ are considered for the following reason. If $c_k(k+\gamma) = 0$ for some γ, the constraint would hold and robot k can enter node p only when robot $k+\gamma$, which is coupled to robot k, enters the succeeding node q. This would result in an infeasible optimization problem. The constraints under these conditions ensure a safe distance between the robots around the input nodes.

4-3-4 Additional constraints on control variables

In the previous sections, many constraints were formulated that put restrictions on the states $x_i(k)$, using max-plus binary control variables. The control variables need to be properly constrained as well, since not all combinations of binary values result in a feasible solution. The constraints apply to conventional binary variables for computational purposes.

Remember that a max-plus binary variable w is active when its value is 0 and inactive when ε. The corresponding conventional binary variable $w^\flat$ is also active when it has value 0, and inactive when 1, due to the relation $w = \beta w^\flat$ with β a very large negative number.

Paths, segments and edges

Each robot is allowed to choose one path. Equation 4-46 makes sure that precisely one path variable takes the value 0, and all others take value 1.

$$\sum_{\ell=1}^{L} w_\ell^\flat(k) = L - 1 \tag{4-46}$$

Note that not all paths are available for a robot, since each robot gets a target node where it needs to deliver its parcel to and not all paths necessarily contain that target node. Denote the set of unavailable paths for robot k by $L_u(k) = \{\ell |$ target of robot $k \notin w_\ell\}$. Each path variable of the excluded paths gets value 1, since it should be inactive. This leads to the constraint for robot k:

$$\sum_{\ell \in L_u(k)} w_\ell^\flat(k) = |L_u(k)| \tag{4-47}$$

To ensure that edge (i, j) is only active when the edge is on the path that robot k travels, the constraint from Equation 4-48 is added, where $L_{i,j} = \{\ell |\ (i, j) \in w_\ell\}$ is the set of paths that include edge (i, j). The edge variable is equal to 1 when none of the paths that include the edge are traveled by robot k, and 0 otherwise.

$$s_{i,j}^\flat(k) = \sum_{\ell \in L_{i,j}} w_\ell^\flat(k) - |L_{i,j}| + 1 \tag{4-48}$$

Lastly, the segment variables $p_l(k)$ need to be constrained to the correct path, such that a segment can only be used if it is actually part of the path that is traveled by robot k. This is done in Equation 4-49, where $L_l = \{\ell |\ p_l \in w_\ell\}$ is the set of paths that include segment l.

$$p_l^\flat(k) = \sum_{\ell \in L_l} w_\ell^\flat(k) - |L_l| + 1 \tag{4-49}$$

Following

The following variables $f_{l,\mu}(k - \mu)$ are max-plus binary control variables, but they do not need to be controlled. They are fixed due to dependencies on other variables and the order in which robots enter the system. The variables are defined for three different cases, when segment l starts at:

1. a node with one incoming edge:
   $f_{l,\mu}^\flat(k - \mu) = f_{\pi_s(l),\mu}^\flat(k - \mu)$
2. a node i with multiple incoming edges:
   $f_{l,\mu}^\flat(k - \mu) = z_{i,\mu}^\flat(k - \mu)$
3. an input node:
   $f_{l,\mu}^\flat(k - \mu) = \begin{cases} 0, & \text{if } \mu > 0 \\ 1, & \text{if } \mu < 0 \end{cases}$

Case 1 ensures that robots keep the same order on segments in succession, since overtaking is not allowed. The constraint in case 2 keeps the chosen order in which robots entered the first node of the segment for the rest of the segment, and case 3 defines the order of robots on the entire segment in the same order as they entered the system.

Coupling

When robot k returns to an input node, it gets a new index $k+m$ due to the coupling variables. The coupling variables need to be constrained in two ways. First, constraints are needed to allow precisely one robot to be coupled to a certain robot $k+m$:

$$\sum_{\gamma=1}^{\gamma_{\max}} c^{\flat}_{k+m-\gamma}(k+m) = \gamma_{\max} - 1 \tag{4-50}$$

Secondly, each robot should be coupled precisely once to a new robot:

$$\sum_{\gamma=1}^{\gamma_{\max}} c^{\flat}_{k}(k+\gamma) = \gamma_{\max} - 1 \tag{4-51}$$

Inputs

All paths end in nodes that precede inputs, and therefore edges to input nodes can not be activated yet. Robots that finish their path need to continue driving to an input to receive their next parcel. Therefore, activating these edges should be made possible. For each node p that precedes an input node, the paths that end in node p are known. The edges to input nodes q are constrained as follows, where $L_p = \{\ell |\ p \text{ is the last node of } w_\ell\}$ is the set of paths that end in node p:

$$\sum_{\substack{q\in\sigma(p)\\ q\in Q}} s^{\flat}_{p,q}(k) = \sum_{\substack{q\in\sigma(p)\\ q\in Q}} 1 + \sum_{\ell\in L_p} w^{\flat}_{\ell}(k) - |L_p| \tag{4-52}$$

On the other hand, robots should travel precisely one edge that starts in an input node. Equation 4-53 forces this on the edge variables.

$$\sum_{q\in Q}\sum_{j\in\sigma(q)} s^{\flat}_{q,j}(k) = \sum_{q\in Q}(|\sigma(q)|) - 1 \tag{4-53}$$

In addition, the input node that robot k drives back to, should be the starting node of the robot it is coupled to. Equation 4-53 already restricts the amount of active edges starting in input nodes to one, but an additional constraint is needed to ensure the right connection. The number of edges starting from a specific input node that can be active is bounded from above for each $q \in Q$ and $\gamma = 1, \ldots, \gamma_{\max}$ as follows:

$$\sum_{j\in\sigma(q)} s^{\flat}_{q,j}(k) \le |\sigma(q)| + c^{\flat}_{k-\gamma}(k) - |\pi(q)| + \sum_{p\in\pi(q)} s^{\flat}_{p,q}(k-\gamma) \tag{4-54}$$

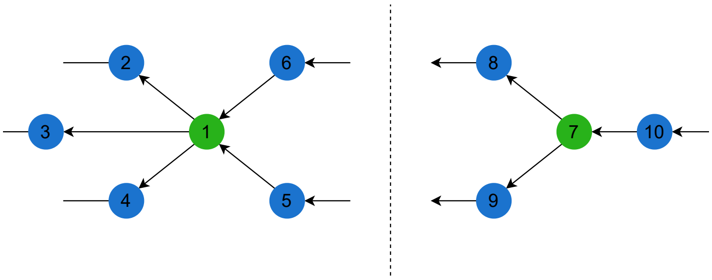


Figure 4-12: Hypothetical input layout.

To illustrate, Figure 4-12 shows the effect of these constraints. Nodes 1 and 7 are input nodes of the same graph, with different amounts of incoming and outgoing edges.

For example, say that robot $k-4$ enters node 1 via node 5, and gets coupled to robot k. For both input nodes, Equation 4-54 becomes

$$\sum_{j\in\sigma(1)} s^{\flat}_{1,j}(k) \leq 3 + c^{\flat}_{k-\gamma}(k) - 2 + \sum_{p\in\pi(1)} s^{\flat}_{p,1}(k-\gamma) \tag{4-55}$$

$$\sum_{j\in\sigma(7)} s^{\flat}_{7,j}(k) \leq 2 + c^{\flat}_{k-\gamma}(k) - 1 + \sum_{p\in\pi(7)} s^{\flat}_{p,7}(k-\gamma) \tag{4-56}$$

For $\gamma = 4$, substitute $c^{\flat}_{k-4}(k) = 0$, $s^{\flat}_{5,1}(k-4) = 0$ and all other edge variables as 1:

$$\sum_{j\in\sigma(1)} s^{\flat}_{1,j}(k) \leq 2 \tag{4-57}$$

$$\sum_{j\in\sigma(7)} s^{\flat}_{7,j}(k) \leq 2 \tag{4-58}$$

Since input node 1 has three outgoing edges, one of them is forced to be 0 (active) for robot k, because of Equation 4-57. The inequality from Equation 4-58 implies that also one of the edges from input node 7 could be active, but the constraint from Equation 4-53 prevents this, since only one edge from an input can be active. For $\gamma = 1, \ldots, \gamma_{\max}, \gamma \neq 4$, the constraints become

$$\sum_{j\in\sigma(1)} s^{\flat}_{1,j}(k) \leq 3 + 1 - 2 + \sum_{p\in\pi(1)} s^{\flat}_{p,1}(k-\gamma) \tag{4-59}$$

$$\sum_{j\in\sigma(7)} s^{\flat}_{7,j}(k) \leq 2 + 1 - 1 + \sum_{p\in\pi(7)} s^{\flat}_{p,7}(k-\gamma) \tag{4-60}$$

The sum over all incoming edge variables for input nodes on the right-hand side is for each γ either equal to the amount of incoming edges, or one less. Then the constraints become either

$$\sum_{j\in\sigma(1)} s^{\flat}_{1,j}(k) \leq 3+1-2+2 \quad \Longrightarrow \quad \sum_{j\in\sigma(1)} s^{\flat}_{1,j}(k) \leq 4 \tag{4-61}$$

$$\sum_{j\in\sigma(7)} s^{\flat}_{7,j}(k) \leq 2+1-1+0 \quad \Longrightarrow \quad \sum_{j\in\sigma(7)} s^{\flat}_{7,j}(k) \leq 2 \tag{4-62}$$

or

$$\sum_{j\in\sigma(1)} s^{\flat}_{1,j}(k) \leq 3+1-2+1 \quad \Longrightarrow \quad \sum_{j\in\sigma(1)} s^{\flat}_{1,j}(k) \leq 3 \tag{4-63}$$

$$\sum_{j\in\sigma(7)} s^{\flat}_{7,j}(k) \leq 2+1-1+1 \quad \Longrightarrow \quad \sum_{j\in\sigma(7)} s^{\flat}_{7,j}(k) \leq 3 \tag{4-64}$$

Robots that do not get coupled to robot k therefore do not influence the enabling of the edges from input nodes for robot k.

Chapter 5

Scheduling

The model that describes the dynamics of the sorting system is complete, taking into account all possible routes, orders and synchronizations. The optimal choices for all robots in the sorting area to minimize the total time for sorting a large amount of parcels with given targets can be found with scheduling. A well-known and widely used scheduling technique is model predictive scheduling (MPS). This chapter discusses theory on MPS and gives an overview of the optimization structure, and how the optimization problem is implemented into the solver Gurobi [14].

5-1 Model predictive scheduling

One of the most popular control methods is model predictive control (MPC). It is popular for its ability to anticipate on future events, its flexibility with respect to disturbances and delays and the possibility of constraining both inputs and outputs. A common approach in MPC is the receding horizon principle. Each time step, optimal inputs are based on future predictions over a certain horizon and available information on the past steps. Only the input for the current step is applied to the system that gives optimal results for the entire horizon, and the horizon shifts to the next time step. This is done each time step, which makes it possible to act on any disturbances. This methodology can be extended to scheduling, hence the term model predictive scheduling [5, 15, 16].

5-1-1 MPS for SMPL systems

In many cases, the elements in switching max-plus linear (SMPL) systems that need to be optimized are binary control variables instead of the input $u(k)$. In the sorting system, only an uncontrollable, external input $u_e(k)$ is present. The control vector consists of all binary decision variables, which are stacked into control vector $v(k)$:

$$v(k) = \begin{bmatrix} w_\ell(k) \\ s_{i,j}(k) \\ f_{l,\mu}(k-\mu) \\ p_l(k) \\ z_{i,\mu}(k-\mu) \\ b_{i,\mu}(k-\mu) \\ c_k(k+\gamma) \end{bmatrix} \tag{5-1}$$

The total SMPL system of the sorting system for robot k can be summarized as

$$x(k) \geq \bigoplus_{\mu=-\mu_{\max}}^{\mu_{\max}} A_\mu(v(k)) \otimes x(k-\mu) \oplus B(v(k)) \otimes u_e(k) \tag{5-2}$$

Introducing the prediction horizon N_p, Equation 5-2 transforms into the following structure:

$$\begin{aligned} \tilde{x}(k) \geq & \tilde{A}_0(v(k)) \otimes \tilde{x}(k) \oplus \tilde{A}_1(v(k)) \otimes x(k-1) \oplus \cdots \oplus \\ & \tilde{A}_{\mu_{\max}}(v(k)) \otimes x(k-\mu_{\max}) \oplus \tilde{B}(v(k)) \otimes \tilde{u}_e(k) \end{aligned} \tag{5-3}$$

where

$$\tilde{x}(k) = \begin{bmatrix} \hat{x}(k) \\ \hat{x}(k+1) \\ \vdots \\ \hat{x}(k+N_p-1) \end{bmatrix}, \quad \tilde{u}_e(k) = \begin{bmatrix} u_e(k) \\ u_e(k+1) \\ \vdots \\ u_e(k+N_p-1) \end{bmatrix} \tag{5-4}$$

with $\hat{x}(k+m)$ the predicted state for $m = 0, \ldots, N_p - 1$. $\tilde{A}_0(v(k))$ is defined as

$$\begin{bmatrix} A_0(0) & \cdots & A_{-\mu_{\max}}(0) & \mathcal{E}_{n\times n} & \cdots & \mathcal{E}_{n\times n} \\ \vdots & \ddots & \vdots & & & \vdots \\ A_{\mu_{\max}}(\mu_{\max}) & \cdots & A_0(\mu_{\max}) & A_{-1}(\mu_{\max}) & \cdots & A_{\mu_{\max}-N_p+1}(\mu_{\max}) \\ \mathcal{E}_{n\times n} & A_{\mu_{\max}}(\mu_{\max}+1) & \cdots & A_0(\mu_{\max}+1) & \cdots & A_{\mu_{\max}-N_p+2}(\mu_{\max}+1) \\ \vdots & & \ddots & \vdots & \ddots & \vdots \\ \mathcal{E}_{n\times n} & \mathcal{E}_{n\times n} & \cdots & A_{N_p-\mu_{\max}-2}(N_p-1) & \cdots & A_0(N_p-1) \end{bmatrix} \tag{5-5}$$

Inside $\tilde{A}_0(v(k))$, the variables $v(k+m)$ are replaced by m in all matrices for brevity. Each of the matrices $A_\mu(m)$ defines relations between robot $k+m$ and $k+m-\mu$. Lastly,

$$\tilde{A}_1(v(k)) = \begin{bmatrix} A_1(v(k)) \\ A_2(v(k+1)) \\ \vdots \\ A_{\mu_{\max}}(v(k+\mu_{\max}-1)) \\ \mathcal{E}_{n\times n} \\ \vdots \\ \mathcal{E}_{n\times n} \end{bmatrix}, \quad \ldots \quad, \tilde{A}_{\mu_{\max}}(v(k)) = \begin{bmatrix} A_{\mu_{\max}}(v(k)) \\ \mathcal{E}_{n\times n} \\ \vdots \\ \mathcal{E}_{n\times n} \end{bmatrix}, \tag{5-6}$$

$$\tilde{B}(v(k)) = \begin{bmatrix} B(v(k)) & \mathcal{E}_{n\times n_u} & \cdots & \mathcal{E}_{n\times n_u} \\ \mathcal{E}_{n\times n_u} & B(v(k+1)) & & \vdots \\ \vdots & & \ddots & \mathcal{E}_{n\times n_u} \\ \mathcal{E}_{n\times n_u} & \cdots & \mathcal{E}_{n\times n_u} & B(v(k+N_p-1)) \end{bmatrix} \tag{5-7}$$

The goal is to find the optimal control sequence $\tilde{v}(k) = (v(k), \ldots, v(k+N_p-1))$ that minimizes the time needed to deliver all parcels in the prediction horizon N_p. In this case, the receding horizon principle applies the optimal decision variables that are still allowed to change, instead of the current input only. With a certain objective function $J(k)$, the complete MPS problem can be formulated for $m = 0, \ldots, N_p - 1$ as

$$\begin{aligned} \min_{\tilde{x}(k), \tilde{v}(k)} \quad & J(k) \\ \text{subject to} \quad & x(k+m) \geq \bigoplus_{\mu=-\mu_e}^{\mu_{\max}} A_\mu(v(k+m)) \otimes x(k+m-\mu) \oplus B(v(k+m)) \otimes u_e(k+m) \\ & A_x(k)\tilde{x}(k) = c_x(k) \\ & A_v(k)\tilde{v}(k) = c_v(k) \end{aligned} \tag{5-8}$$

For cycles nearing the end of the prediction horizon, there are no constraints to look $\mu_{\max}$ cycles ahead. This is why μ_e is defined as $\min(\mu_{\max}, N_p - 1 - m)$. States and binary decision variables that are not allowed to change are fixed with the constraints in $A_x(k)$ and $A_v(k)$ respectively. Which decision variables need to be fixed, depends on what information is available at the time instant at which the optimization is done.

Timing issues

In SMPL systems, state changes are initiated by events and the states represent event times which are often easy to measure. Therefore, the timing in MPS is different from MPC for conventional continuous-time systems. The MPS problem is still solved at time each instant t, but since the event counter k is not related to a specific time, it is not directly known which event is the current event. At a certain time instant t, information on the states for current and previous events is needed to solve the MPS problem. Define the current event k as

$$k = \max\Big\{\kappa \mid x_i(\kappa-1) \leq t, \ \forall i \in \{1, \ldots, n\}\Big\} \tag{5-9}$$

This means that for the optimization at time instant t, all elements of the state vector $x(k-1)$ are fully measured and known [5]. In the sorting system, this means that robot $k-1$ has finished its path, and robot k has not yet. It is possible for the states of robot k and future robots to be partially known, since they already entered the system and have visited some nodes. Therefore, also some binary decision variables such as traveled edges and orders between a pair of robots on certain segments are already known. These are not allowed to change anymore, but the optimization for unknown decision variables is done, based on the available information. The horizon shifts when robot k has finished its route.

5-1-2 Optimization problem

The optimization in the MPS problem results in an optimal time schedule for the robots through the sorting area, making routing, ordering and synchronization choices for each robot in the prediction horizon N_p. At each time step t, this optimization is done for the prediction horizon starting at cycle k. All information on the states of cycles $k-1, k-2, \ldots, k-\mu_{\max}$ is known and cannot be changed. The result is an optimal schedule for cycles $k, \ldots, k+N_p-1$.

A type of optimization problem for which many solvers are available is called mixed integer linear programming (MILP) problem. In an MILP problem, the objective function and constraints are linear in the optimization vector, which consists of both real-valued and integer-valued parameters. MILP problems are in general NP-hard, which means that computation time of an MILP problem grows exponentially with the size of the problem [17]. Nevertheless, there are many fast and reliable solvers available that are able to solve these types of problems, such as CPLEX, Gurobi or XPRESS [18]. Gurobi is used to implement the MILP problem for the sorting system. The implementation is shown in section B-1.

A general framework for MILP problems is given in Equation 5-10, where $\zeta(k)$ is the optimization vector. The MPS problem from Equation 5-8 consists of constraints that are linear in the max-plus algebra. Since they consist of maximization and addition only, they can be converted to inequalities in the conventional algebra. The MPS problem can therefore be recast into an MILP problem, if also the objective function is max-plus linear.

$$\begin{aligned}
&\min_{\zeta(k)} c^T \zeta(k) \\
\text{subject to } & H\zeta(k) \leq b \\
&\zeta(k) = \begin{bmatrix} \zeta_r(k) \\ \zeta_i(k) \end{bmatrix}, b \in \{\theta(k), b_0(k), 0, \beta\} \\
&\zeta_r(k) \in \mathbb{R}^{n_r}, \zeta_i(k) \in \mathbb{Z}^{n_i}
\end{aligned} \tag{5-10}$$

The optimization vector $\zeta(k)$ consists of two main parts. The first is $\zeta_r(k)$, containing all real-valued variables, which are the current and future states for the prediction horizon $(x(k), \ldots, x(k+N_p-1))$. The second part $\zeta_i(k)$ contains all integer-valued variables, which are the binary control variables for the prediction horizon $(v(k), \ldots, v(k+N_p-1))$. Vector b contains linear combinations of the travel times of each edge from past, current and future robots in $\theta(k)$, all known states and binary control variables that cannot be changed in $b_0(k)$ and potentially the scalar β. In some cases, an element of b is equal to 0.

The objective function, the constraint matrix H on the left-hand side and the vector on the right-hand side of the inequality will be discussed separately. But first, an overview of the vectors in the optimization problem is given.

Structure

The optimization vector $\zeta(k)$ and the known vectors $\theta(k)$ and $b_0(k)$ are filled with many variables. The structure of these vectors as implemented for the optimization in Python is shown below. Define P as the graph representing a floor plan.

$$\zeta_r(k) = \begin{bmatrix} x(k) \\ x(k+1) \\ \vdots \\ x(k+N_p-1) \end{bmatrix}, \quad \zeta_i(k) = \begin{bmatrix} R \\ O \\ S \end{bmatrix}, \quad \theta(k) = \begin{bmatrix} T \\ T_p \end{bmatrix}, \quad b_0(k) = \begin{bmatrix} X_p \\ R_p \\ O_p \\ S_p \\ I \end{bmatrix} \tag{5-11}$$

where R, O and S refer to binary control variables for routing, ordering and synchronizations respectively, T and T_p contain travel times of all edges for future and past cycles respectively, and I refers to the input times. The subscript p indicates that the variables from that set are from previous cycles, of which all information is known. For $\zeta_i(k)$, the structure is as follows:

$$R = \begin{bmatrix} s_{i,j}(k) \\ \vdots \\ s_{i,j}(k+N_p-1) \\ w_\ell(k) \\ \vdots \\ w_\ell(k+N_p-1) \end{bmatrix} \quad {\scriptstyle \substack{\forall (i,j)\in\mathcal{D}(P), \\ \ell=1,\dots,L}}$$

$$O = \begin{bmatrix} f_{l,\mu}(k) \\ \vdots \\ f_{l,\mu}(k+N_p-1) \\ p_l(k) \\ \vdots \\ p_l(k+N_p-1) \\ z_{i,\mu}(k) \\ \vdots \\ z_{i,\mu}(k+N_p-1) \end{bmatrix} \quad {\scriptstyle \substack{\forall l=1,\dots,L_s, \\ \mu=1,\dots,\mu_{\max}, \\ i\in\mathcal{N}_{\text{ord}}}} \qquad S = \begin{bmatrix} b_{i,\mu}(k) \\ \vdots \\ b_{i,\mu}(k+N_p-1) \\ c_{k+\gamma_1}(k+r) \\ c_{k+\gamma_2}(k+r+1) \\ \vdots \\ c_{k+\gamma_p}(k+N_p-1) \end{bmatrix} \quad {\scriptstyle \substack{\forall i\in\mathcal{N}_{\text{syn}}, \\ \mu=1,\dots,\mu_{\max}, \\ \gamma_1=0,\dots,r-1, \\ \gamma_2=0,\dots,r, \\ \vdots \\ \gamma_p=N_p-1-\gamma_{\max},\dots,N_p-2}}$$

The structure of the coupling vectors is a bit more complicated than the others. How the coupling variables are stored will be discussed at the end of this paragraph. For $\theta(k)$, the structure is

$$T = \begin{bmatrix} \tau_{i,j}(k) \\ \vdots \\ \tau_{i,j}(k+N_p-1) \end{bmatrix} \quad {\scriptstyle \forall (i,j)\in\mathcal{D}(P)}, \qquad T_p = \begin{bmatrix} \tau_{i,j}(k-1) \\ \vdots \\ \tau_{i,j}(k-\mu_{\max}) \end{bmatrix} \quad {\scriptstyle \forall (i,j)\in\mathcal{D}(P)}$$

And lastly, $b_0(k)$ is built up like

$$X_p = \begin{bmatrix} x(k-1) \\ \vdots \\ x(k-\mu_{\max}) \end{bmatrix}, \qquad R_p = \begin{bmatrix} s_{i,j}(k-1) \\ \vdots \\ s_{i,j}(k-\mu_{\max}) \\ w_\ell(k-1) \\ \vdots \\ w_\ell(k-\mu_{\max}) \end{bmatrix} \; \substack{\forall (i,j)\in\mathcal{D}(P), \\ \ell=1,\ldots,L}$$

$$O_p = \begin{bmatrix} f_{l,\mu}(k-1) \\ \vdots \\ f_{l,\mu}(k-\mu_{\max}) \\ p_l(k-1) \\ \vdots \\ p_l(k-\mu_{\max}) \\ z_{i,\mu}(k-1) \\ \vdots \\ z_{i,\mu}(k-\mu_{\max}) \end{bmatrix} \; \substack{\forall l=1,\ldots,L_s, \\ \mu=1,\ldots,\mu_{\max}, \\ i\in\mathcal{N}_{\mathrm{ord}}}, \qquad S_p = \begin{bmatrix} b_{i,\mu}(k-1) \\ \vdots \\ b_{i,\mu}(k-\mu_{\max}) \\ c_{k-\gamma_1}(k) \\ \vdots \\ c_{k-\gamma_g}(k+\gamma_{\max}-1) \end{bmatrix} \; \substack{\forall i\in\mathcal{N}_{\mathrm{syn}}, \\ \mu=1,\ldots,\mu_{\max}, \\ \gamma_1=1,\ldots,\gamma_{\max}, \\ \vdots \\ \gamma_g=1}$$

$$I = \begin{bmatrix} u_e(k) \\ \vdots \\ u_e(k+N_p-1) \end{bmatrix}$$

Regarding the coupling variables, the structure is not as straightforward as for the other types of variables. Other variables are known when the index is in the past, that is for $k-\mu$ with $\mu > 0$. This is also the case for coupling variables, but it also holds that coupling variables with indices in the future ($k-\mu$ with $\mu < 0$) can still be known. This is due to the active, driving robots which already have been coupled. This holds for all $k, \ldots, k+r-1$, which are the active robots. On the other hand, for robots $k+\gamma_{\max}, \ldots, k+N_p-1$, all coupling variables are unknown and need to be determined in the optimization problem. The robots in between, robots $k+r, \ldots, k+\gamma_{\max}-1$, have a few known and a few unknown variables. From now on, set $\gamma_{\max} = \mu_{\max}$, such that only the previous robots that are already taken into account in the optimization can be coupled to a new robot.

For clarification on the structure of the optimization vector and known vector containing the coupling variables, the reader is referred to Equation 5-12. To keep the implementation of the constraints structured, the lengths of the parts of the vectors regarding one index are all $\gamma_{\max}$. For indices $k+r, \ldots, k+\gamma_{\max}-1$, part of the coupling constraints are known, and others are still to be optimized. Therefore, there are not $\gamma_{\max}$ variables to fill the part of the vector. The remainder of that part of the vector is replaced with 0.5, to imply that there is no variable there. The braces at the sides show which variables belong to each index.

$$\begin{array}{cc}
\text{unknown (in } S) & \text{known (in } S_p) \\
\begin{array}{r}
\gamma_{\max}\left\{\begin{matrix} c_{k+r-1}(k+r) \\ \vdots \\ c_k(k+r) \\ 0.5 \\ \vdots \\ 0.5 \end{matrix}\right. \\
\gamma_{\max}\left\{\begin{matrix} c_{k+r}(k+r+1) \\ \vdots \\ c_k(k+r+1) \\ 0.5 \\ \vdots \\ 0.5 \end{matrix}\right. \\
\vdots \\
\gamma_{\max}\left\{\begin{matrix} c_{k+\gamma_{\max}-1}(k+\gamma_{\max}) \\ \vdots \\ c_k(k+\gamma_{\max}) \end{matrix}\right. \\
\vdots \\
\gamma_{\max}\left\{\begin{matrix} c_{k+N_p-2}(k+N_p-1) \\ \vdots \\ c_{k+N_p-1-\gamma_{\max}}(k+N_p-1) \end{matrix}\right.
\end{array}
&
\begin{array}{l}
\left.\begin{matrix} c_{k-1}(k) \\ \vdots \\ c_{k-\gamma_{\max}}(k) \end{matrix}\right\}\gamma_{\max} \\
\vdots \\
\left.\begin{matrix} c_{k+r-2}(k+r-1) \\ \vdots \\ c_{k+r-1-\gamma_{\max}}(k+r-1) \end{matrix}\right\}\gamma_{\max} \\
\left.\begin{matrix} 0.5 \\ \vdots \\ 0.5 \\ c_{k-1}(k+r) \\ \vdots \\ c_{k+r-\gamma_{\max}}(k+r) \end{matrix}\right\}\gamma_{\max} \\
\left.\begin{matrix} 0.5 \\ \vdots \\ 0.5 \\ c_{k-1}(k+r+1) \\ \vdots \\ c_{k+r+1-\gamma_{\max}}(k+r+1) \end{matrix}\right\}\gamma_{\max} \\
\vdots \\
\left.\begin{matrix} 0.5 \\ \vdots \\ 0.5 \\ c_{k-1}(k+\gamma_{\max}-1) \end{matrix}\right\}\gamma_{\max}
\end{array}
\end{array} \tag{5-12}$$

With the introduction of the prediction horizon, the additional constraints on the coupling variables for cycles nearing the end of the horizon need some adjustment. Remember the additional constraint that ensures that a robot gets coupled to precisely one new robot:

$$\sum_{\gamma=1}^{\gamma_{\max}} c_k^\flat(k+\gamma) = \gamma_{\max} - 1 \tag{5-13}$$

This constraint holds for robots $k, \ldots, k+N_p-1-\gamma_{\max}$, since for each of these robots, the range of robots they can be coupled to is inside the prediction horizon. For later robots, looking $\gamma_{\max}$ robots ahead is outside the scope of the prediction horizon. It can happen that a robot is coupled to none of the robots ahead and in the prediction horizon. Therefore, the equality constraint from Equation 5-13 would raise an infeasible model for robots $k' = k+N_p-\gamma_{\max}, \ldots, k+N_p-1$. Define $\gamma_{\text{end}} = k+N_p-1-k'$. For robots k', the additional coupling constraints become

$$\sum_{\gamma=1}^{\gamma_{\text{end}}} c_{k'}^\flat(k'+\gamma) \geq \gamma_{\text{end}} - 1 \tag{5-14}$$

Objective function

The optimization minimizes a certain objective function $J(k)$ which is a linear function of the optimization vector: $J(k) = c^T\zeta(k)$. The objective function that is defined for the sorting system consists of two parts. The first part, $J_x(k)$, minimizes the state values, and the second part, $J_u(k)$, puts a penalty on driving back to the wrong input.

In the sorting system, the goal is to sort all parcels in the prediction horizon as fast as possible. Therefore, the objective function wants to minimize the largest state value of each of the robots $k, \dots, k+N_p-1$. In other words, a robot should enter the last node on its path at the smallest time instant possible. In addition, the robots should enter the nodes as soon as they are able to, thus minimizing each state variable. This objective is less important than returning to an input as fast as possible, therefore they are multiplied by a small scalar value. The objectives are captured in Equation 5-15.

$$J_x(k) = \sum_{m=0}^{N_p-1} \max_{i=1,\dots,n} x_i(k+m) + \sum_{m=0}^{N_p-1} \sum_{i=1}^{n} 0.01 x_i(k+m) \tag{5-15}$$

In the solver Gurobi, the first part of this objective function is implemented by introducing a new optimization variable `max_x`. The objective is to minimize `max_x`, while its value is being pushed towards the highest event time by adding the following constraints for $i = 1, \dots, n$ and robots $k, \dots, k+N_p-1$:

$$x_i(k) \leq \texttt{max\_x} \tag{5-16}$$

It is also desired to have robots drive back to the input where the next parcel is expected to be available for a pickup. Remember that a conventional binary variable equals 0 when it is active, and 1 when inactive. By penalizing the use of edges starting in the desirable input node, the minimization will force an edge from that input node to be active. The expected input nodes for the entire prediction horizon are sorted in time and denoted by q_m for robot $k+m$. All edges starting in the input node to be used by the robots are then added to the objective function as follows:

$$J_u(k) = \sum_{m=0}^{N_p-1} \sum_{j \in \sigma(q_m)} 100 s^{\flat}_{q_m,j}(k+m) \tag{5-17}$$

The final linear objective function is defined in Equation 5-18, which is a linear combination of elements from the optimization vector $\zeta(k)$.

$$J(k) = J_x(k) + J_u(k) \tag{5-18}$$

Constraint matrix

The optimization problem from Equation 5-10 defines a set of linear constraints with respect to optimization vector $\zeta(k)$. These constraints are contained in matrix H. In chapter 4, all constraints to model the sorting system are given, first in max-plus algebra and then in conventional algebra. Remember the max-plus linear routing constraints for node i:

$$x_i(k) \geq \bigoplus_{j \in \pi(i)} x_j(k) \otimes \tau_{j,i}(k) \otimes s_{j,i}(k) \tag{5-19}$$

In conventional algebra, this becomes

$$x_i(k) \geq \max_{j \in \pi(i)} x_j(k) + \tau_{j,i}(k) + \beta s^\flat_{j,i}(k) \tag{5-20}$$

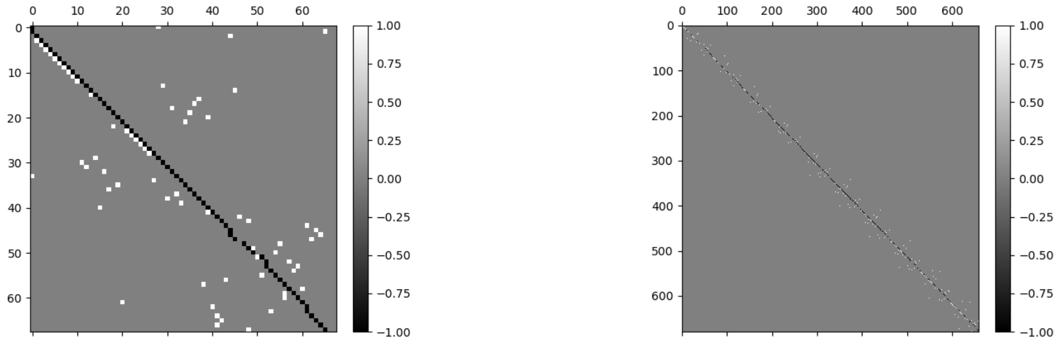
This is essentially a set of inequalities. The states are larger or equal to the maximum of a set of linear functions, so the states should be larger than each of the linear functions separately, as was already shown in Equation 4-5 and 4-6:

$$\begin{aligned} x_i(k) &\geq x_{j_1}(k) + \tau_{j_1,i}(k) + \beta s^\flat_{j_1,i}(k) \\ &\vdots \\ x_i(k) &\geq x_{j_p}(k) + \tau_{j_p,i}(k) + \beta s^\flat_{j_p,i}(k) \end{aligned} \tag{5-21}$$

Here, p is the amount of predecessors to node i. All types of constraints that are defined for the model can be written into this structure. Therefore, the constraints can be implemented into the MILP problem formulation $H\zeta(k) \leq b$, where each row in H represents one constraint. To make the inequalities less or equal, the constraints need to be multiplied by -1. All variables that are in the optimization vector are situated on the left-hand side of the equation. The resulting MILP formulation of the routing constraints therefore becomes

$$\begin{aligned} x_{j_1}(k) - x_i(k) + \beta s^\flat_{j_1,i}(k) &\leq -\tau_{j_1,i}(k) \\ &\vdots \\ x_{j_p}(k) - x_i(k) + \beta s^\flat_{j_p,i}(k) &\leq -\tau_{j_p,i}(k) \end{aligned} \tag{5-22}$$

The values in the constraint matrix H are all in the set $\{\beta, -1, 0, 1, -\beta\}$ with respect to the correct indices in the optimization vector $\zeta(k)$. H is the collection of all constraints in the model for cycles $k, \ldots, k + N_p - 1$. In Python, a part of matrix H is created for each type of constraints, divided over the variables that appear in the constraints. For the routing constraints, this means that parts of H are created each with the amount of rows equal to the amount of routing constraints. The first part is `H_x` with respect to the states x, which has a 1 in column j_1 and -1 in column i for the first row. The amount of columns is the amount of states, times the prediction horizon. In addition, `H_s` is made, which contains the constraints with respect to edge variables $s_{i,j}$. It has β in the column that corresponds to $s_{j_1,i}(k)$ in the first row. The amount of columns is equal to the amount of edges in the graph, multiplied by the prediction horizon. To illustrate, the structure of `H_x` is shown in Figure 5-1. In Figure 5-1a, all routing constraints with respect to x for one robot are shown, where each row has an element equal to 1 and to -1. Figure 5-1b shows the routing constraints for x for a prediction horizon of $N_p = 10$ robots. This is a repetition of the routing constraint matrix for one robot on the diagonal, since routing constraints are the same for each robot individually.



(a) Routing constraints for one robot.

(b) Routing constraints for 10 robots.

Figure 5-1: Structure of the routing constraints with respect to state variable x.

Index matrix

The vector b is on the right-hand side of inequalities in MILP format, where each element is a linear combination of known variables for one constraint. For routing constraints, this is only one parameter, $-\tau_{j,i}(k)$. For other types of constraints, the right-hand side can consist of the sum of multiple parameters and even β when the adjoint of a variable is taken. In other cases, there might be no elements on the right-hand side at all. Take one set of ordering constraints from Equation 4-32 where robot k is compared to robots $k-\mu$ and $k+\mu$:

$$\begin{aligned} x_i(k) &\geq x_j(k-\mu) + \beta z^{\flat}_{i,\mu}(k-\mu) + \beta p^{\flat}_l(k) + \beta p^{\flat}_m(k-\mu) \\ x_i(k) &\geq x_j(k+\mu) + \beta(1 - z^{\flat}_{i,\mu}(k)) + \beta p^{\flat}_l(k+\mu) + \beta p^{\flat}_m(k) \end{aligned} \tag{5-23}$$

The ordering constraints can be recast into the MILP format as follows:

$$\begin{aligned} -x_i(k) + \beta p^{\flat}_l(k) \;&\leq\; -x_j(k-\mu) - \beta z^{\flat}_{i,\mu}(k-\mu) - \beta p^{\flat}_m(k-\mu) \\ -x_i(k) + x_j(k+\mu) - \beta z^{\flat}_{i,\mu}(k) + \beta p^{\flat}_l(k+\mu) + \beta p^{\flat}_m(k) \;&\leq\; -\beta \end{aligned} \tag{5-24}$$

Note that the right-hand side of the constraints can indeed consist of multiple variables and the scalar β. In other cases, the right-hand side is equal to 0. Therefore, b is constructed as follows:

$$\begin{aligned} b &= F \rightarrow b^*(k) - s \\ b^*(k) &= \begin{bmatrix} 0 \\ \theta(k) \\ b_0(k) \end{bmatrix}, \; [s]_i \in \{0, \beta\} \; \forall i \end{aligned} \tag{5-25}$$

A matrix-vector multiplication with a large, almost empty matrix is computationally heavy. This can be avoided by choosing F to be an index matrix, implicated by the → operator [6].

This is a matrix consisting of pointers to the elements in $b^*(k)$. If no binary decision variables are on the right-hand side of a constraint, the index matrix points to the first element of b, which is 0. The scalar vector s takes care of scalar additions due to adjoint variables in the constraint.

To illustrate, the right-hand side of Equation 5-24 is created with this structure in Equation 5-26. Indexing starts at 0, since the model is implemented in Python. The integers x, z and p point to the correct locations in $b^*(k)$ with respect to the state and binary variables $-x_j(k-\mu)$, $-\beta z^{\flat}_{i,\mu}(k-\mu)$, $-\beta p^{\flat}_m(k-\mu)$ respectively. Their locations can be deduced from the structure of $b_0(k)$, described before in paragraph Structure.

$$b = \begin{bmatrix} x & z & p \\ 0 & 0 & 0 \end{bmatrix} \to b^*(k) - \begin{bmatrix} 0 \\ \beta \end{bmatrix} \tag{5-26}$$

Constraints for a robot that take into account other robots, such as the ordering and synchronization constraints, result in a more complex constraint matrix than the routing constraints considering only one robot. Consider again the ordering constraints from Equation 5-24. The following example creates the constraint and index matrices for ordering with respect to the state variable x. For every node that has an ordering variable assigned to it, constraints are added for robot k with respect to $k-1$, $k+1$, $k-2$, $k+2, \dots, k-\mu_{\max}$, $k+\mu_{\max}$ in that order. The constraint matrix can be restructured such that all constraints regarding a single robot $k-\mu$ are grouped together. This results in the matrix H_0 that gets multiplied with robot k, where each row has one non-zero element equal to -1, and matrices H_μ for robot $k-\mu$. In Equation 5-27, the constraint and index matrices are shown, where *0* are matrices of appropriate dimensions containing only zeros.

$$\left[\begin{array}{ccccccccc}
H_0 & H_1^+ & \mathit{0} & \cdots & & & & \cdots & \mathit{0} \\
H_0 & \mathit{0} & H_2^+ & \mathit{0} & \cdots & & & \cdots & \mathit{0} \\
\vdots & \vdots & \ddots & \ddots & \ddots & & & & \vdots \\
H_0 & \mathit{0} & \cdots & \mathit{0} & H_{\mu_{\max}}^+ & \mathit{0} & \cdots & \cdots & \mathit{0} \\
\hline
H_1 & H_0 & H_1^+ & \mathit{0} & \cdots & & & \cdots & \mathit{0} \\
\mathit{0} & H_0 & \mathit{0} & H_2^+ & \mathit{0} & \cdots & & \cdots & \mathit{0} \\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots & & & \vdots \\
\mathit{0} & H_0 & \mathit{0} & \cdots & \mathit{0} & H_{\mu_{\max}}^+ & \mathit{0} & \cdots & \mathit{0} \\
\hline
\mathit{0} & H_1 & H_0 & H_1^+ & \mathit{0} & \cdots & & \cdots & \mathit{0} \\
H_2 & \mathit{0} & H_0 & \mathit{0} & H_2^+ & \mathit{0} & \cdots & \cdots & \mathit{0} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \ddots & & & \vdots \\
\mathit{0} & \mathit{0} & H_0 & \mathit{0} & \cdots & \mathit{0} & H_{\mu_{\max}}^+ & & \mathit{0} \\
\hline
 & & & & \vdots & & & & \\
 & & & & \vdots & & & & \\
\hline
\mathit{0} & \cdots & & & & \cdots & \mathit{0} & H_1 & H_0 \\
\mathit{0} & \cdots & & & \cdots & \mathit{0} & H_2 & \mathit{0} & H_0 \\
\vdots & & & & \iddots & \iddots & \iddots & \vdots & \vdots \\
\mathit{0} & \cdots & \cdots & \mathit{0} & H_{\mu_{\max}} & \mathit{0} & \cdots & \mathit{0} & H_0
\end{array}\right]
\begin{bmatrix} x(k) \\ x(k+1) \\ x(k+2) \\ \vdots \\ x(k+N_p-1) \end{bmatrix}
\leq
\left[\begin{array}{c}
F_1 \\ F_2 \\ \vdots \\ F_{\mu_{\max}} \\ \hline \mathit{0} \\ F_1 \\ \vdots \\ F_{\mu_{\max}-1} \\ \hline \mathit{0} \\ \mathit{0} \\ \vdots \\ F_{\mu_{\max}-2} \\ \hline \vdots \\ \vdots \\ \hline \mathit{0} \\ \mathit{0} \\ \vdots \\ \mathit{0}
\end{array}\right]
\to
\begin{bmatrix} 0 \\ x(k-1) \\ x(k-2) \\ \vdots \\ x(k-\mu_{\max}) \end{bmatrix} \tag{5-27}$$

The matrices H_μ are identical for $\mu = 1, \dots, \mu_{\max}$, where all odd rows have a non-zero element equal to 1 and thus construct the part of a constraint with respect to x for robot $k-\mu$, and all even rows consist of zeros. The matrices H_μ^+ represent the disturbed constraints with respect to robots $k+\mu$ and they are the same as H_μ, but the even and odd rows are switched. For example, all nodes in the considered sorting area that have an ordering variable are the nodes with multiple incoming arcs. These are nodes 0, 44, 52 and 61. Matrix H_0 is built as follows, where only the non-zero elements are shown:

$$H_0 = \begin{array}{ccccccccc} 0 & \cdots & 44 & \cdots & 52 & \cdots & 61 & \cdots & 65 \\ \hline -1 & & & & & & & & \\ -1 & & & & & & & & \\ & & -1 & & & & & & \\ & & -1 & & & & & & \\ & & & & -1 & & & & \\ & & & & -1 & & & & \\ & & & & & & -1 & & \\ & & & & & & -1 & & \end{array} \tag{5-28}$$

Since the ordering constraints consider nodes with multiple incoming arcs and their successor nodes, matrices H_μ have non-zero elements for the successor nodes, which are 32, 1, 50 and 43 respectively. For all $\mu = 1, \ldots, \mu_{\max}$, the constraint matrices for $k \pm \mu$ therefore have the following form:

$$H_\mu = \begin{array}{ccccccccccc} 0 & 1 & \cdots & 32 & \cdots & 43 & \cdots & 50 & \cdots & 65 \\ \hline & & & 1 & & & & & & \\ & & & & & & & & & \\ & 1 & & & & & & & & \\ & & & & & & & & & \\ & & & & & & & 1 & & \\ & & & & & & & & & \\ & & & & & 1 & & & & \\ & & & & & & & & & \end{array}, \quad H_\mu^+ = \begin{array}{cccccccccc} 0 & 1 & \cdots & 32 & \cdots & 43 & \cdots & 50 & \cdots & 65 \\ \hline & & & & & & & & & \\ & & & 1 & & & & & & \\ & & & & & & & & & \\ & 1 & & & & & & & & \\ & & & & & & & & & \\ & & & & & & & 1 & & \\ & & & & & & & & & \\ & & & & & 1 & & & & \end{array} \tag{5-29}$$

Index matrices F_μ are used as pointers to states of robots that already finished, where no constraint matrix H_μ is present. The index matrices have non-zero elements on the odd rows only. To complete the example, the structure of F_μ is as follows:

$$F_\mu = \begin{bmatrix} 32 + 66(\mu - 1) \\ 0 \\ 1 + 66(\mu - 1) \\ 0 \\ 50 + 66(\mu - 1) \\ 0 \\ 43 + 66(\mu - 1) \\ 0 \end{bmatrix} \tag{5-30}$$

Note that wherever an index matrix has a 0, the corresponding constraint matrix has a non-zero element in that row. The other way around also holds: where an index matrix has a non-zero element, the corresponding constraint matrix has a row with zeros.

5-2 Online optimization

The optimization problem from Equation 5-10 is implemented in Python by constructing all the constraint matrices and pointer matrices for each type of constraint and each variable that appears in the constraints. This scheduling problem is solved by Gurobi for robots $k, \ldots, k+N_p-1$. But when the sorting system is running, this optimization needs to be done online, taking into account available information at the moment of optimization. The online optimization with a receding horizon approach consists of three steps. At each time step, the model needs to be updated with real-time information on the system, the optimization of the updated model is executed, and finally the optimal choices are passed on to the robots in the system. An overview of the online optimization is given in Figure 5-2. Due to time restrictions, the receding horizon strategy is not implemented. This section describes an idea for implementation of the online optimization.

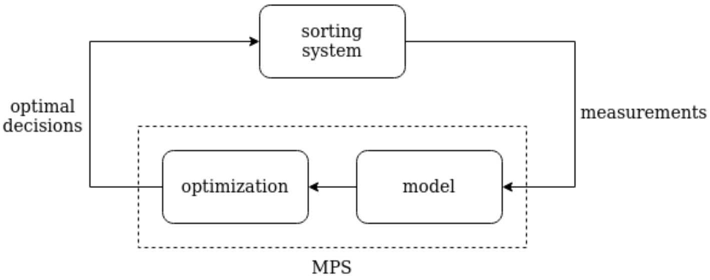


Figure 5-2: Online optimization structure.

5-2-1 Update model

In online optimization, the scheduling problem is solved for robots in the prediction horizon at each time step t, with a fixed time interval between consecutive time steps. At each time step, real-time information on the system needs to be updated in the SMPL model. The states represent time instances, which can be measured precisely. Since some of the states of the robots in the prediction horizon are not allowed to change anymore, since the robots have already passed these nodes, they need to be fixed with equality constraints. The same needs to be done for all decision variables that can not change anymore.

For example, let robot k pick up a parcel at input node 46, and let it at time step t be somewhere between node 44 and node 1, as shown in Figure 5-3. There are certain variables that need to be fixed. Robot k has traveled path $(46, 41, 64, 44)$. Therefore, states $x_{46}(k)$, $x_{41}(k)$, $x_{64}(k)$ and $x_{44}(k)$ are known and should be fixed. Also, the traveled edges are known and the decision variables representing them should not be able to change: $s^{\flat}_{46,41}(k) = s^{\flat}_{41,64}(k) = s^{\flat}_{64,44}(k) = s^{\flat}_{44,1}(k) = 0$. The travel times that robot k needed to travel these edges until node 44 can also be measured and substituted into $\theta(k)$ at the right-hand side of the MILP problem. Moreover, node 44 has two incoming arcs. Since robot k has passed this node, the order in which k entered node 44 in relation to robots $k \pm \mu$ for $\mu = 1, \ldots, \mu_{\max}$ is also known and should be fixed. At certain points in the graph, information on segments $(p_l(k))$ and paths $(w_\ell(k))$ is also known. All known information can be included in the model by adding equality constraints as defined in the MPS problem in Equation 5-8.

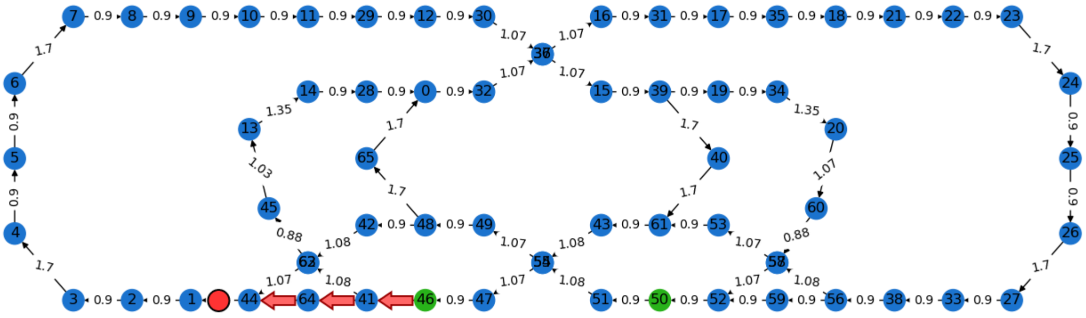


Figure 5-3: Some edge variables are fixed at time t, indicated by a red arrow.

As mentioned earlier, there is a constant number of r robots driving through the sorting area, and the prediction horizon N_p is taken larger than r. At the start of the sorting process ($t = 0$) the horizon consists of robots $[0, \ldots, N_p - 1]$ and the active robots are $[0, \ldots, r - 1]$. Both lists of robots have a fixed length of N_p and r respectively. When a robot finishes its job and returns to an input node, it is no longer an active robot. It gets removed from the list of active robots and robots in the horizon, and added to the list with finished robots. The horizon is extended until it has N_p elements, and the list of active robots until it has r elements. Note that the finished robot is coupled to this new active robot with a coupling variable.

The list of active robots is an ascending series, and if at time step $t > 0$ the difference between the first and last elements is exactly $r - 1$, the structure of the problem is the same as for $t = 0$. It is not a given that the robot at the beginning of the active list is also the first robot to return. If another robot finishes first, the structure of the MILP problem changes. Therefore, two types of updates are distinguished. The horizon shifts when the first robot finishes its job, and the constraint matrices need to be adjusted when another robot finishes. Assume that the list of active robots at time step $t - 1$ is $[k, \ldots, k + r - 1]$.

Shift horizon

It is important to keep track of the currently active robots and the latest robots that finished in order to keep the model up to date. When at time step t, robot k has finished its job, it gets removed from the list of active robots and added to the list of finished robots. This operation is shown in Equation 5-31.

$$\begin{array}{ccc} \text{active} & \qquad & \text{finished} \\ \begin{bmatrix} k \\ \vdots \\ k + r - 1 \end{bmatrix} \Rightarrow \begin{bmatrix} k + 1 \\ \vdots \\ k + r \end{bmatrix} & \qquad & \begin{bmatrix} k - 1 \\ \vdots \\ k - \mu_{\max} \end{bmatrix} \Rightarrow \begin{bmatrix} k \\ \vdots \\ k - \mu_{\max} + 1 \end{bmatrix} \end{array} \tag{5-31}$$

The horizon shifts in the same way as the active robots, and becomes $[k + 1, \ldots, k + N_p]$. It is possible that multiple robots have finished between the time instants $t - 1$ and t. If the robots are all at the beginning of the list of active robots, both lists should shift with the amount of robots that finished. The structure of the scheduling problem does not change in this case, and the optimization can be executed again. If, on the other hand, a later robot finishes before the first, the structure of the constraint matrices does change.

Adapt constraint matrices

When a robot has finished at time t that is not the first active robot, say robot $k+1$, the lists change in the following way:

$$\begin{array}{cc} \text{active} & \text{finished} \\ \begin{bmatrix} k \\ \vdots \\ k+r-1 \end{bmatrix} \Rightarrow \begin{bmatrix} k \\ k+2 \\ k+3 \\ \vdots \\ k+r \end{bmatrix} & \begin{bmatrix} k-1 \\ \vdots \\ k-\mu_{\max} \end{bmatrix} \Rightarrow \begin{bmatrix} k-1 \\ \vdots \\ k-\mu_{\max} \\ k+1 \end{bmatrix} \end{array} \tag{5-32}$$

Note that the horizon does not shift, since robot k is still active, and that the length of the finished list is variable. The robot that finished gets added to that list, but no robots are removed from the list. This is because robot k is still active, and the constraints for k with respect to $k-\mu$ should not be neglected. Because of this change in structure in the future and past vectors, the structure of the constraint matrices also changes. This is not the case for constraint matrices for routing constraints, since these constraints for robot k do not take into account other robots. Ordering and synchronization constraints, on the other hand, are created with respect to other robots, so those constraint matrices need to be adapted to fit the active robots.

Remember the structure of the ordering constraint matrix with respect to state variable x from Equation 5-27. This structure changes due to the change in active and finished robot list structures. The changes are shown in bold in Equation 5-33. Note that the amount of constraints stays constant.

Because robot $k+1$ already finished and is moved to the right-hand side of the equation, the constraints regarding robot $k+1$ do not end up in the constraint matrix, but in the index matrix. Two types of adaptations are distinguished. The first one applies to robots in the prediction horizon that started earlier than the already finished robot, and the second one applies to robots that started later than the finished robot.

When a robot (k) has constraints with respect to a later robot that already finished ($k+1$), the constraints for the whole prediction horizon remain unchanged, except for the finished robot. This is done by replacing H_1^+ with a matrix containing only zeros. Note that in Equation 5-33, the constraints regarding the remaining part of the prediction horizon are shifted upwards. Since H_μ^+ are the same for all $\mu = 1, \ldots, \mu_{\max}$, the only notable change in the constraint matrix is adding zeros instead of H_1^+. In the index matrix, the same shift upwards takes place, and the rows where zeros are added in the constraint matrix, pointers are added in the index matrix. This is denoted with an asterisk. Each constraint has either a non-zero element in the constraint matrix or in the index matrix. In the nominal case, zeros and non-zeros alternate rows. But in the case where a later robot finishes earlier, this changes. The pointers in F_1^* do not only contain the constraints regarding $x(k-1)$, but on the other rows, they point to states of robot $k+1$, which are added to the bottom of the previous state vector. F_1^* is shown in Equation 5-34.

$$\left[\begin{array}{cccccccccc}
H_0 & \boldsymbol{H_2^+} & 0 & \cdots & & & & & \cdots & 0 \\
H_0 & 0 & \boldsymbol{H_3^+} & 0 & \cdots & & & & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \ddots & & & & & \vdots \\
H_0 & 0 & \cdots & 0 & \boldsymbol{H^+_{\mu_{\max}}} & 0 & \cdots & & \cdots & 0 \\
H_0 & 0 & \cdots & & \cdots & \mathbf{0} & \cdots & & \cdots & 0 \\
\hline
\mathbf{0} & H_0 & H_1^+ & 0 & \cdots & & & & \cdots & 0 \\
\boldsymbol{H_2} & H_0 & 0 & H_2^+ & 0 & \cdots & & & \cdots & 0 \\
0 & H_0 & 0 & 0 & H_3^+ & 0 & \cdots & & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & & & \vdots \\
0 & H_0 & 0 & \cdots & \cdots & 0 & H^+_{\mu_{\max}} & 0 & \cdots & 0 \\
\hline
0 & H_1 & H_0 & H_1^+ & 0 & \cdots & & & \cdots & 0 \\
\mathbf{0} & 0 & H_0 & 0 & H_2^+ & 0 & \cdots & & \cdots & 0 \\
\boldsymbol{H_3} & 0 & H_0 & 0 & 0 & H_3^+ & & & & 0 \\
\vdots & \vdots & \vdots & \vdots & & \ddots & \ddots & & & \vdots \\
0 & 0 & H_0 & 0 & \cdots & \cdots & 0 & H^+_{\mu_{\max}} & & 0 \\
\hline
 & & & & & \vdots & & & & \\
 & & & & & \vdots & & & & \\
\hline
0 & \cdots & & & & & \cdots & 0 & H_1 & H_0 \\
0 & \cdots & & & & \cdots & 0 & H_2 & 0 & H_0 \\
\vdots & & & & & \iddots & \iddots & \iddots & \vdots & \vdots \\
0 & \cdots & & \cdots & 0 & H_{\mu_{\max}} & 0 & \cdots & 0 & H_0
\end{array}\right]
\begin{bmatrix} x(k) \\ x(k+2) \\ x(k+3) \\ \vdots \\ x(k+N_p) \end{bmatrix}
\leq
\left[\begin{array}{c}
\boldsymbol{F_2} \\ \boldsymbol{F_3} \\ \vdots \\ \boldsymbol{F_{\mu_{\max}}} \\ \boldsymbol{F_1^*} \\
\hline
\mathbf{0^*} \\ \mathbf{0} \\ \boldsymbol{F_1} \\ \vdots \\ \boldsymbol{F_{\mu_{\max}-2}} \\
\hline
0 \\ \mathbf{0^*} \\ \mathbf{0} \\ \vdots \\ \boldsymbol{F_{\mu_{\max}-3}} \\
\hline
\vdots \\ \vdots \\
\hline
0 \\ 0 \\ \vdots \\ 0
\end{array}\right]
\rightarrow
\begin{bmatrix} 0 \\ x(k-1) \\ x(k-2) \\ \vdots \\ x(k-\mu_{\max}) \\ x(k+1) \end{bmatrix}
\tag{5-33}$$

When a robot $(k+1)$ finishes before the first robot in the prediction horizon (k), the constraint structure of the robots that started later than the finished robot also needs adaptations. Consider the constraints for robot $k+2$ in the second block of constraints in the constraint matrix. The first row defines constraints with respect to robots $k+2 \pm \mu$ for $\mu = 1$, so for robots $k+1$ and $k+3$. The constraints in the constraint matrix with respect to robot $k+1$ get substituted by an all-zero matrix, and added to the index matrix. The index matrix was originally a zero vector, but now it becomes 0^*, which is defined in Equation 5-34.

$$F_1^* = \begin{bmatrix} 32 \\ 32+66\mu_{\max} \\ 1 \\ 1+66\mu_{\max} \\ 50 \\ 50+66\mu_{\max} \\ 43 \\ 43+66\mu_{\max} \end{bmatrix}, \quad 0^* = \begin{bmatrix} 0 \\ 32+66\mu_{\max} \\ 0 \\ 1+66\mu_{\max} \\ 0 \\ 50+66\mu_{\max} \\ 0 \\ 43+66\mu_{\max} \end{bmatrix} \tag{5-34}$$

The second row of constraints $(\mu = 2)$ for robot $k+2$ take into account robots k and $k+4$. Therefore, both H_2 and H_2^+ are present in the constraint matrix, and the index matrix becomes zero. The result is that for robots that started later than the finished robots, the index matrix shifts downwards and 0^* is added above. A setup is made for the implementation of MPS in section B-2, where a start is made for the adaptations of the constraint and index matrices.

The adaptations to the matrices of the MILP problem are different for different types of constraints. For routing constraints that only apply to a single robot k, nothing changes. This also holds for the additional constraints defined in subsection 4-3-4. For constraints built for robots k and $k \pm \mu$ like ordering and synchronization constraints, the matrices are subjected to systematic adaptations as previously described. The last type of constraints is coupling, which has a different structure than ordering and synchronization constraints since it relates robot k to robots $k - \gamma$ only for $\gamma > 0$. The structural changes of the optimization matrices are comparable to the changes explained for future and past robots, but only taking into account the past robots. Note that the vectors containing the coupling variables are built differently than vectors of other variables, as shown in Equation 5-12.

5-2-2 Update system

After the model update, possibly including a shift in the horizon and adaptations to the constraint matrices, the SMPL system is optimized. This results in an optimal time schedule for the robots, which can be derived from the states $x(k), \ldots, x(k + N_p - 1)$, and the corresponding optimal choices, represented by all binary variables in the model. To make online optimization possible, these choices need to be processed and communicated to the robots in the sorting system. Due to time restrictions, the optimal choices are not implemented into the system.

An idea to optimize the sorting process is to use the choices in edges, orders and synchronizations to control the robots. The important variables to use are $s_{i,j}(k)$, $z_{i,\mu}(k - \mu)$ and $b_{i,\mu}(k - \mu)$ for all robots. Including the optimal choices for ordering in nodes with multiple incoming edges and for synchronizations creates a new way of path planning for this system.

A brief explanation on the current path planning was given in chapter 2. Robots claim nodes, which become available to claim for other robots once that robot has left the node. The example from Figure 2-3 showed that the claiming method used now does not always prove to be optimal. Therefore, the orders and synchronizations in practice might differ from the optimal orders and synchronizations according to the SMPL model of the system. The optimal decision variables can be used in a certain way to force the orders in which robots claim nodes into the system. Details on the implementation plan are left out for confidentiality reasons. What follows is an explanation of the way the optimized decision variables for synchronizations are stored and how they should be interpreted. This is similar for the ordering variables.

The optimized synchronization variables are shown in Figure 5-4. Each cell represents a synchronization variable $b^{\flat}_{i,\mu}(k)$, which defines the order between robots k and $k + \mu$. Each row contains the variables for robot k for all nodes with a synchronization variable and all $\mu = 1, \ldots, \mu_{\max} = 9$. There are four locations in the sorting area that need a synchronization variable, which are nodes $i = 41, 56, 30, 43$ in the graph. The columns are grouped per μ, and contain the variables for all those nodes.

The highlighted part defines the order between robot 0 and 1, if there is a synchronization needed between the two robots. The variable $b^{\flat}_{41,1}(0) = 0$, meaning that robot 0 goes before robot 1 in the synchronization at node 41. The synchronizations for robot 0 with respect to robot 1 on the other nodes are all equal to 1. This either means that robot 0 has to wait for robot 1 on these nodes, or that no synchronization is needed. This is the case when the

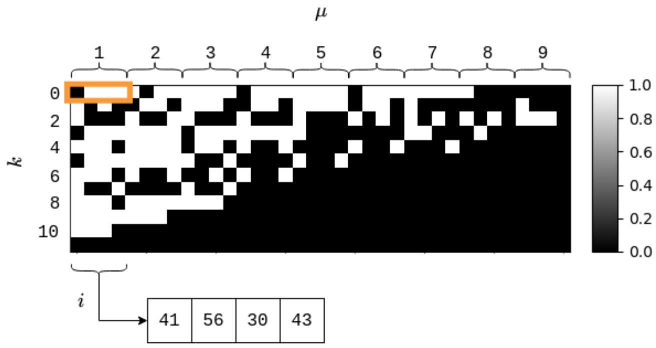


Figure 5-4: Synchronization variables $b^{\flat}_{i,\mu}(k)$ after running an optimization.

robots are traveling the same segment where the synchronization takes place, and thus follow each other, or when one of the robots does not travel past that synchronization point.

Chapter 6

Visualization

To test the model accuracy and to see the robots navigate through the sorting area according to their optimal decisions, the results should be visualized. This is done through a simulation in Python with the help of the `matplotlib` library. This chapter discusses a simulation and highlights important moments where robots make certain choices.

The chosen simulation has 8 active robots (r) and a prediction horizon N_p of 12, which means that in total 4 robots are coupled to a new robot and receive a second parcel. At most 8 robots are driving around at the same time, and in total 12 parcels get sorted. Each robot takes into account 9 robots that started earlier and later both for ordering and synchronization ($\mu_{\max}$) and for coupling ($\gamma_{\max}$). The target nodes are chosen randomly, and the input times are drawn from the gamma distribution defined in subsection 4-2-3 for each input and scaled for confidentiality reasons. The input times, input nodes and target nodes for all robots are presented in Table 6-1.

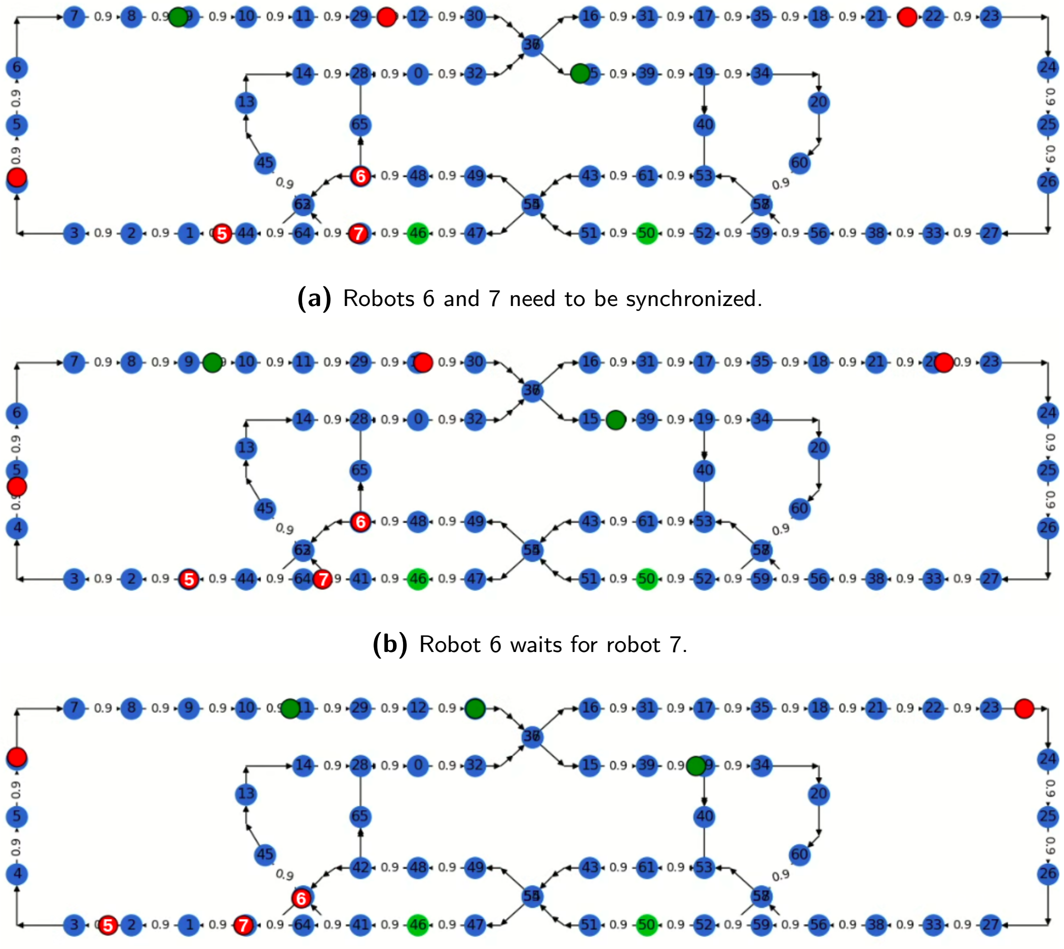
k	u_e	Input q	Target
0	1.0	46	29
1	2.0	50	30
2	7.5	50	27
3	9.2	46	7
4	17.5	50	12
5	17.8	46	30
6	20.6	50	29
7	25.6	46	8
8	26.0	50	27
9	30.1	46	22
10	30.9	50	35
11	33.1	46	26

Table 6-1: External factors for the simulation with $r = 8$, $\mu_{\max} = \gamma_{\max} = 9$ and $N_p = 12$.

The figures in this chapter are screenshots taken from the simulation. Robots are red when they are carrying a parcel and become green when they arrive at their target. There is a notable difference between the way the edges are depicted in the screenshots and in the figures of the floor plan shown before. In the earlier figures, it was important to show which nodes are connected to each other. In the sorting system, robots drive via certain points. Therefore in the simulation the edges are drawn accordingly. The code for the visualization is given in Appendix C.

Ordering and synchronization

The decisions for ordering and synchronization were previously explained separately. In the used sorting area, all nodes with multiple incoming edges (ordering) are also near synchronization configurations where nodes are too close to each other. Therefore, the choices have much overlap in the simulation and they are discussed together. Figure 6-1 shows a situation where two robots travel to node 44 at the bottom left through different edges and an order needs to be chosen. In addition, the preceding nodes 62 and 64 are not allowed to be occupied simultaneously due to the small distance between them.



(a) Robots 6 and 7 need to be synchronized.

(b) Robot 6 waits for robot 7.

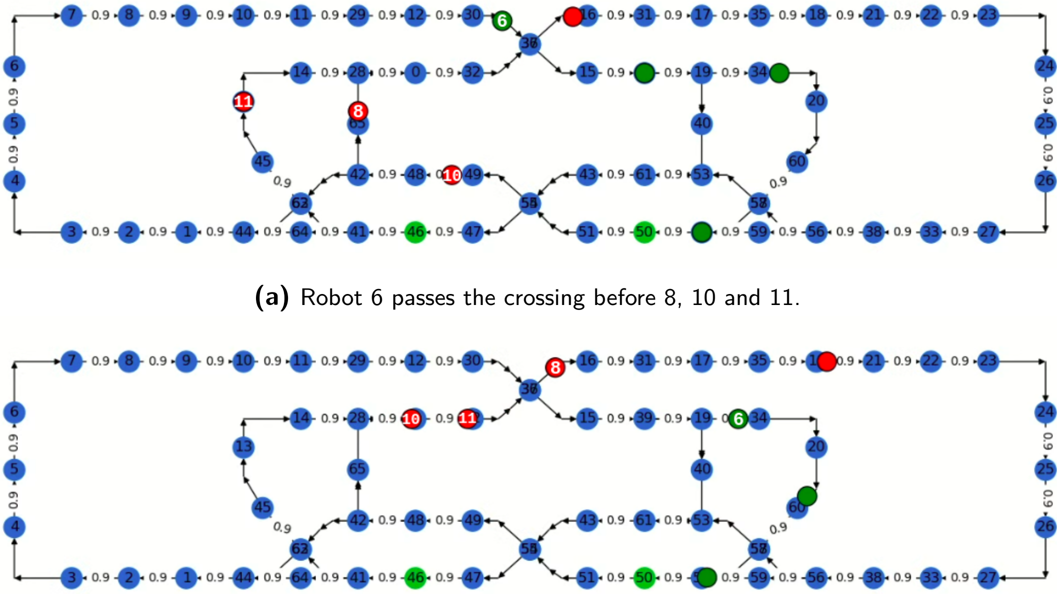
(c) The order between the robots in node 44 is the same as for the synchronization.

Figure 6-1: Example of two robots switching order.

The choice is made to let robot 7 go ahead of robot 6 in the synchronization configuration for node 42, where the decision variable is $b_{42,1}(6) = \varepsilon$. This means that robot 6 has to stay in its position until node 64, the node too close to the robot's successor node, has been left by robot 7 as shown in Figure 6-1a and 6-1b. As a consequence, the order in node 44 is the same: $z_{44,1}(6) = \varepsilon$. This is depicted in Figure 6-1c, where a safe distance is maintained between the robots. Since robot 6 started its job earlier than robot 7, this synchronization enabled a switch in order between robots.

Delays

In the same simulation, another synchronization is made at the crossing at the top. This simulation is a nominal case, assuming no delays in the system. Adding a delay to an edge shows the adaptability of the model. In the nominal case, the order of robots 6, 8, 10 and 11 passing the crossing is shown in Figure 6-2. Robot 6 crosses first, and robots 8, 11, 10 follow in this order.



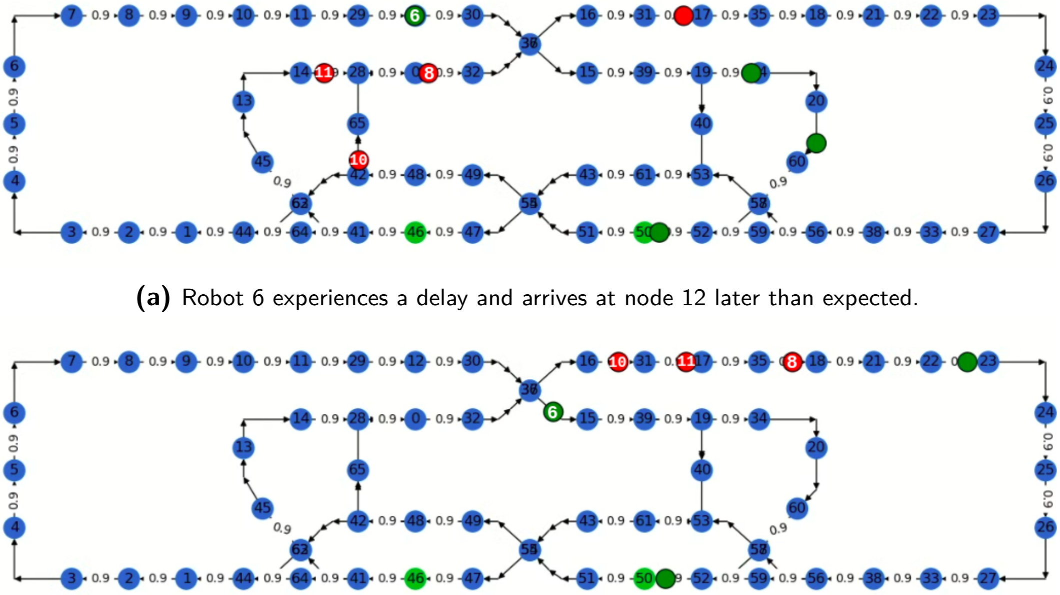
(a) Robot 6 passes the crossing before 8, 10 and 11.

(b) Robots 8, 11 and 10 are the next robots to cross.

Figure 6-2: A synchronization without any delays.

The target node for robot 6 is 29. It is possible that the robot takes a longer time to drop the parcel off than expected. Where in the previous case, robot 6 goes ahead of the other robots, it might be optimal to change this order if the drop-off takes more time than expected. A delay of 4 seconds is added to edge $(29, 12)$ for robot 6, which results in a different optimal schedule. Figure 6-3 shows the same situation as before, but robot 6 is running behind. Because robot 6 arrives at the crossing later, the optimization indeed returns that robots 8, 11 and 10 should go ahead of robot 6.

The switch in order can also be seen in the synchronization variables, highlighted in Figure 6-4. The node for the synchronization variables at the considered crossing is $i = 30$, and

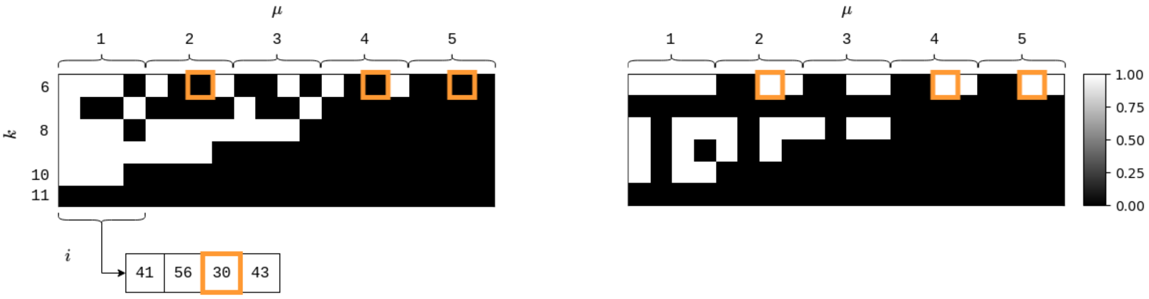


(a) Robot 6 experiences a delay and arrives at node 12 later than expected.

(b) Because of the delay, robots 8, 11 and 10 pass the crossing before robot 6.

Figure 6-3: The same synchronization with a delay.

the variables are shown for robot 6 in relation to robots 8, 10 and 11. In the nominal case in Figure 6-4a, robot 6 passes the crossing before the other three robots, and therefore $b^{\flat}_{30,\mu}(6) = 0$ for $\mu = 2, 4, 5$. In the case where robot 6 is delayed, the same variables become equal to 1, as shown in Figure 6-4b. This matches with the visualization, where robot 6 passes the crossing later than the other three robots.



(a) Nominal case.

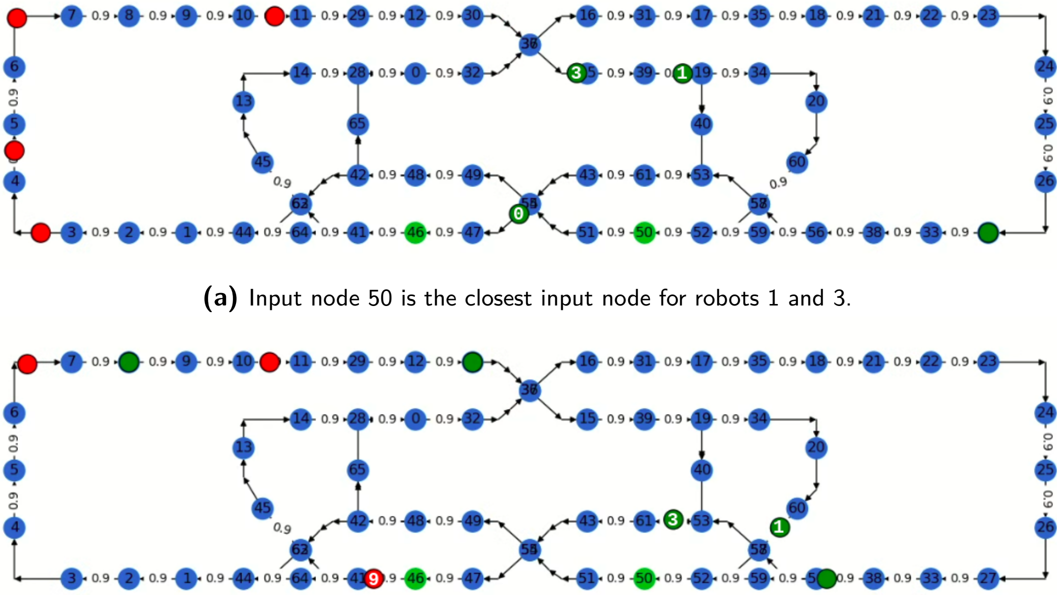
(b) Robot 6 is delayed.

Figure 6-4: Synchronization variables $b^{\flat}_{30,\mu}$ change due to the delay.

Adding a delay on an edge shows that the model is able to adapt to the situation. This is a useful characteristic of the model when implementing the online optimization, since at each time step the model is updated with newly gained information, which might differ from the situation predicted by the model predictive scheduling (MPS) model.

Input choice and coupling

Part of the objective function in the optimization problem is assigned to minimizing the waiting time for operators at the input nodes. When a robot has dropped off its parcel, it chooses to return to an input node to receive its next parcel, taking into account the availability of parcels at the input nodes. Figure 6-5a shows robot 1 and 3 both driving back to an input. The path to input node 50 is shorter than to input node 46, so the robots would want to drive back to node 50. But because there is a parcel available at node 46 as well, they choose different input nodes, as shown in Figure 6-5b. Both figures also show an example of coupling. Robot 0 returns at input node 46 where it gets is next parcel, and continues on its next job as robot 9.



(a) Input node 50 is the closest input node for robots 1 and 3.

(b) Robots 1 and 3 choose different inputs. Robot 0 gets coupled to robot 9.

Figure 6-5: Robots choose different inputs to minimize operator waiting time, and at an input node they get coupled to a new robot.

The switching max-plus linear (SMPL) model for the autonomous sorting system is made to fit any sorting area in which all nodes can be reached from any other node. The visualization has been successfully tested on multiple floor plans that are in use at Prime Vision. These floor plans need to stay confidential, so the results are not shown here.

Chapter 7

Conclusion

The focus of this thesis research was to create a switching max-plus linear (SMPL) model that represents the Autonomous Sorting system at the software company Prime Vision. Creating the model, the optimization and the visualization were the three main topics in this thesis, where the creation of the model answers the main research question:

Can the Autonomous Sorting system at Prime Vision be modeled with an SMPL model?

This question can now be answered with a simple *yes*. The process of achieving this answer and the model will be discussed in this section. In addition, the following sub questions will be answered:

1. *Is it possible to create a flexible SMPL model that can be applied to different sorting areas?*
2. *How can an optimal schedule be obtained from the SMPL model?*
3. *How can the resulting optimal schedule be used to control the system?*

The model has been constructed as follows. Three different types of decisions are distinguished, and system matrices defining the SMPL system for each decision type are constructed for the sorting system. The routing matrix describes the paths that robots can choose from, the ordering matrix defines orders between robots in nodes on merging paths, and the synchronization matrix ensures a safe distance between robots on crossing paths. A general way to construct the system matrices is found, based on the definition of the sorting area that can be represented by a graph consisting of nodes and edges. Decision variables are therein automatically defined for nodes with multiple incoming edges and nodes that are too close to each other, which form the building blocks for the ordering and synchronization matrices respectively. In addition, the paths between all combinations of input nodes and the segments between input nodes and nodes with multiple incoming or outgoing edges are computed. This structural approach makes it possible to create an SMPL model for any sorting area, answering sub question 1 with a *yes*. The only condition is that the sorting area should be represented by a graph that is strongly connected.

The system matrices are used to systematically create max-plus linear constraints on the states of the robots, which form the basis of the model predictive scheduling (MPS) problem. The constraints only contain the operations maximization and addition, which makes it possible to convert them to a set of linear inequalities in the conventional algebra. The resulting optimization problem is a mixed integer linear programming (MILP) problem containing real-valued and integer-valued variables, if an objective function is stated which is also linear. The linear objectives of the sorting system are defined to minimize the time it takes to sort a set of parcels, and to minimize the waiting time of operators at the input nodes. By using an available solver to solve the MILP problem, optimal decisions for all robots in the prediction horizon are found to minimize the two objectives, along with a time schedule for all robots in the sorting area. This concludes the answer to sub question 2.

To provide an answer for sub question 3, a plan for online optimization is proposed which includes a receding horizon approach. It is important that the model gets correctly updated with the latest information on the locations of the robots before the next optimization starts. In the dynamic sorting system, it is possible that robots that started later, finish earlier than other robots, which gives the prediction horizon a different structure. This structurally changes the constraints in the optimization problem as well. A method is proposed to adapt the constraints for an accurate system representation at each optimization step, where certain parts of the constraint matrix and index matrix shift and other parts get replaced by either zeros or pointers. The decisions on order and synchronization choices that result from the optimization can be implemented to the system, potentially resulting in a higher throughput.

To conclude, the constraints and objective functions have been implemented in Python, and the MILP problem is solved with Gurobi. To make an accurate representation of the sorting system, the input times are based on real sorting data, but manipulated for confidentiality reasons. The optimization for a certain prediction horizon, which is equal to the amount of parcels sorted, is visualized through a simulation written in Python. The simulation clearly shows the decisions that robots make and lets the robots wait for other robots when necessary. Adding a distortion on an edge in the floor plan shows the adaptability of the model to disturbances. The model can be applied to any strongly connected sorting area.

Recommendations

Unfortunately, there was not enough time to implement the MPS problem into the sorting system real-time. An implementation plan is presented in chapter 5, which would be interesting to follow up on as a next step.

An important issue that should be kept in mind, is the optimization time. The computation time increases as the floor plans become more complex when they include more nodes and more decision variables. If research shows that the computation takes too long for online optimization, here are a few ideas that could decrease the computation time.

- The coupling variables are needed to schedule and visualize all robots in the prediction horizon, with a smaller amount of active robots. When optimizing every few seconds while the sorting system is running, the coupling variables can be neglected, as they are not important for the navigation of the robots. Only the orders and paths for active robots are of importance. Removing the coupling variables from the model decreases the amount of constraints and binary variables, resulting in lower computational costs.

- Reducing the amount of robots ahead and before a robot that are taken into account ($\mu_{\max}$) could also reduce the computation time. Since robots have the possibility to choose from multiple routes which can differ in length, it can happen that robots driving near each other are more than $\mu_{\max}$ apart if that value is taken too small. A way to keep this value small without running into this problem, is to have a varying $\mu_{\max}$ for certain nodes, depending on nominal differences between robots at that location. Another way is to renumber the robots such that robots driving near each other are inside the $\mu_{\max}$ range.
- To speed up the optimization, the previous solutions require adaptations to the model, but another solution can be found in adapting the MILP problem. The constraint and index matrix can be restructured to make the centralized MPS problem into a distributed model predictive scheduling (DMPS) problem. The system gets partitioned into multiple subsystems, and the constraint matrix is easier for the solver to handle [3].

Before the optimization from the SMPL model gets implemented, it is important to know whether the optimal scheduling achieves a higher throughput. Worst and best case analysis for SMPL systems can be done using the maximum and minimum growth rate respectively. The maximum growth rate is based on random switching and can be solved with a linear programming problem [15, 19], and the minimum growth rate is based on optimal switching and is a lot harder to compute [20]. The throughput of the system can also be determined with Monte Carlo simulation, by running the optimization repeatedly for many different input times and targets [21].

The MILP problem is solved with Gurobi, which is one of the fastest and most reliable solvers available and often used for these types of optimization problems. It would be interesting to see how open source solvers would perform on this model. An open source solver with promising characteristics is MIPCL [22].

Appendix A

Constraints

A-1 Routing constraints

The construction of the routing constraints is shown in this section. Important elements are `H_x`, `H_s` and `pointer_vector`. The first two are the routing constraint matrices with respect to the states x and edge variables $s_{i,j}$ for the prediction horizon `pred_h` respectively. They are created according to Algorithm 1 in the methods `x` and `s`. Method `pointer` creates the index matrix (`pointer_vector`) that points to the correct travel times saved in `tau_indices`.

```python
import numpy as np
import copy

from class_graph import Subgraph
from class_node import InputNode


class Routing:

    def __init__(self, graph, max_plus_model):
        self.variables = copy.deepcopy(max_plus_model.variables["states"])
        for var, num in max_plus_model.variables["routing"].items():
            self.variables.update({var: num})

        self.n_rc = 0

        self.routing_matrix = max_plus_model.routing_matrix
        self.graph = graph
        self.max_plus_model = max_plus_model

        self.H_x = []
        self.H_s = []
        self.pointer_vector = []

    def x(self, pred_h):
        H = np.zeros([1, self.max_plus_model.nx])
        for node_i, i in enumerate(self.routing_matrix):
            if node_i not in self.graph.multi_in and np.any(i) and not isinstance(self.graph.node_dict[node_i], InputNode):
                node_j = np.nonzero(i)          # from node_j to node_i
                new_constraint = np.zeros(self.max_plus_model.nx)
                new_constraint[node_i] = -1
                new_constraint[node_j] = 1
                H = np.vstack([H, new_constraint])
            elif node_i in self.graph.multi_in:
                for arr in np.nonzero(i):
                    for j_index in arr:
                        new_constraint = np.zeros(self.max_plus_model.nx)
                        new_constraint[node_i] = -1
                        new_constraint[j_index] = 1
                        H = np.vstack([H, new_constraint])
        H = H[1:]
        self.n_rc = H.shape[0]        # amount of routing constraints for a single cycle
        self.H_x = np.kron(np.eye(pred_h), H)
        return self.H_x, self.n_rc

    def s(self, pred_h):
        H = np.zeros([self.n_rc, self.max_plus_model.ns])
        for i, constraint in enumerate(self.H_x[:self.n_rc, :self.max_plus_model.nx]):
            s_from = np.where(constraint == 1)
            s_to = np.where(constraint == -1)
            s_index = self.max_plus_model.s_indices["s_" + str(s_from[0][0]) + "_" + str(s_to[0][0])]
            H[i, s_index] = beta
        self.H_s = np.kron(np.eye(pred_h), H)
        return self.H_s

    def pointer(self, pred_h):
        tau_pointer = np.zeros([self.n_rc, 1])
        for i, constraint in enumerate(self.H_x[:self.n_rc, :self.max_plus_model.nx]):
            tau_from = np.where(constraint == 1)
            tau_to = np.where(constraint == -1)
            tau_pointer[i] = self.max_plus_model.tau_indices["tau_" + str(tau_from[0][0]) + "_" + str(tau_to[0][0])] + 1     # +1 because a 0 is added above the theta vector
        self.pointer_vector = np.zeros([pred_h*self.n_rc, 1])
        for i in range(pred_h):
            self.pointer_vector[self.n_rc*i:self.n_rc*(i+1)] = tau_pointer + (i*len(self.max_plus_model.tau_indices))
        return self.pointer_vector


beta = -1000
```

A-2 Ordering constraints

The construction of the ordering constraints is shown in the code below. The important elements are Ho_x, Ho_p and Ho_z, which are the ordering constraint matrices with respect to the states x, segment variables p_l and ordering variables $z_{i,\mu}$ respectively for the whole prediction horizon. They are created according to Algorithm 2 in the methods o_x and o_p and o_z. These methods also create the pointers (index matrices) for ordering constraints with respect to all relevant variables. Lastly, the method o_scalar constructs the scalar vector containing only elements with value 0 or β.

The creation of only the ordering constraints is shown in this class. Similar methods are defined for the construction of following and splitting constraints, but they are left out since the code is repetitive. In those methods, the letter 'o' is replaced by 'f' and 'e' respectively.

```python
import numpy as np
import copy

from class_node import InputNode


class Ordering:

    def __init__(self, graph, max_plus_model):
        self.variables = copy.deepcopy(max_plus_model.variables["states"])
        for var, num in max_plus_model.variables["ordering"].items():
            self.variables.update({var: num})

        self.n_oc = 0            # number of ordering constraints
        self.max_plus_model = max_plus_model
        self.graph = graph

        self.Ho_x = []
        self.Ho_p = []
        self.Ho_z = []
        self.o_x_pointer = []
        self.o_p_pointer = []
        self.o_z_pointer = []
        self.o_scalar_vector = []

    def o_x(self, pred_h, mu_max):
        H = np.zeros([1, self.max_plus_model.nx*(mu_max+1)])
        next_nodes = {}
        pred_segments = {}
        for node in self.graph.multi_in:
            if not isinstance(self.graph.node_dict[node], InputNode):
                row = self.max_plus_model.ordering_matrix[node, :]
                next_nodes[node] = []
                curr_pred_segments = copy.deepcopy(self.graph.multi_in_pred_segments[node])
                pred_segments[node] = curr_pred_segments
                constraint = np.zeros(self.max_plus_model.nx*(mu_max+1))
                constraint[node] = -1
                for array_of_next_nodes in np.nonzero(row):
                    for next_node in array_of_next_nodes:
                        next_nodes[node].append(next_node)
                        for pred_segment in curr_pred_segments:
                            diff_pred_segments = copy.deepcopy(curr_pred_segments)
                            diff_pred_segments.remove(pred_segment)
                            for _ in diff_pred_segments:
                                for mu in range(mu_max):
                                    disturbed_constraint = copy.deepcopy(constraint)
                                    disturbed_constraint[next_node+(mu+1)*self.max_plus_model.nx] = 1
                                    H = np.vstack([H, constraint, disturbed_constraint])
        H = H[1:]
        self.n_oc = H.shape[0]      # amount of ordering constraints for a single cycle

        # repeat matrix for whole prediction horizon
        Ho_x = np.zeros([self.n_oc*pred_h, self.max_plus_model.nx*(pred_h+mu_max)])
        for i in range(pred_h):
            Ho_x[self.n_oc*i:self.n_oc*(i+1), self.max_plus_model.nx*i:self.max_plus_model.nx*(i+1+mu_max)] = H
        self.Ho_x = Ho_x[:, :self.max_plus_model.nx*pred_h]

        # add normal constraints for cycles > k, such that they can switch with earlier robots (filling the lower triangular part of the matrix)
        Ho = np.zeros([1, self.max_plus_model.nx*mu_max])
        for node, list_of_next_nodes in next_nodes.items():
            for next_node in list_of_next_nodes:
                for pred_segment in pred_segments[node]:
                    diff_pred_segments = copy.deepcopy(pred_segments[node])
                    diff_pred_segments.remove(pred_segment)
                    for _ in diff_pred_segments:
                        for mu in range(mu_max):
                            constraint = np.zeros([1, self.max_plus_model.nx*mu_max])
                            constraint[0, next_node + (mu_max-mu-1)*self.max_plus_model.nx] = 1
                            Ho = np.vstack([Ho, constraint, np.zeros([1, self.max_plus_model.nx*mu_max])])
        Ho = Ho[1:]

        for cycle in range(pred_h-1):
            amount = min(mu_max, cycle+1)
            self.Ho_x[self.n_oc*(cycle+1):self.n_oc*(cycle+2), self.max_plus_model.nx*(cycle+1-amount):self.max_plus_model.nx*(cycle+1)] = Ho[:, -self.max_plus_model.nx*amount:]

        # create pointer for ordering constraints with respect to variable x
        px = np.zeros([1, 1])
        for node, list_of_next_nodes in next_nodes.items():
            for next_node in list_of_next_nodes:
                for pred_segment in pred_segments[node]:
                    diff_pred_segments = copy.deepcopy(pred_segments[node])
                    diff_pred_segments.remove(pred_segment)
                    for _ in diff_pred_segments:
                        for mu in range(mu_max):
                            px = np.vstack([px, next_node+self.max_plus_model.nx*mu + 1, 0])
        px = px[1:]
        p = px.shape[0]

        self.o_x_pointer = np.zeros([p*pred_h, 1])
        for cycle in range(mu_max):
            for num in range(int(p/(2*mu_max))):
                self.o_x_pointer[cycle*p + 2*(num*mu_max+cycle):cycle*p + (num+1)*2*mu_max] = px[num*2*mu_max:(num+1)*2*mu_max-2*cycle]
        return self.Ho_x, self.o_x_pointer, self.n_oc

    def o_p(self, pred_h, mu_max):
        H = np.zeros([1, self.max_plus_model.np*(mu_max+1)])
        next_nodes = {}
        pred_segments = {}
        for node in self.graph.multi_in:
            if not isinstance(self.graph.node_dict[node], InputNode):
                row = self.max_plus_model.ordering_matrix[node, :]
                next_nodes[node] = []
                curr_pred_segments = copy.deepcopy(self.graph.multi_in_pred_segments[node])
                pred_segments[node] = curr_pred_segments
                for array_of_next_nodes in np.nonzero(row):
                    for next_node in array_of_next_nodes:
                        next_nodes[node].append(next_node)
                        for pred_segment in curr_pred_segments:
                            pred_index = self.max_plus_model.p_indices["p_" + str(pred_segment)]
                            constraint = np.zeros(self.max_plus_model.np*(mu_max+1))
                            constraint[pred_index] = beta
                            diff_pred_segments = copy.deepcopy(curr_pred_segments)
                            diff_pred_segments.remove(pred_segment)
                            for diff_pred_segment in diff_pred_segments:
                                diff_pred_index = self.max_plus_model.p_indices["p_" + str(diff_pred_segment)]
                                for mu in range(mu_max):
                                    disturbed_constraint = np.zeros(self.max_plus_model.np*(mu_max+1))
                                    disturbed_constraint[diff_pred_index] = beta
                                    disturbed_constraint[pred_index+(mu+1)*self.max_plus_model.np] = beta
                                    H = np.vstack([H, constraint, disturbed_constraint])
        H = H[1:]
        Ho_p = np.zeros([self.n_oc*pred_h, self.max_plus_model.np*(pred_h+mu_max)])
        for i in range(pred_h):
            Ho_p[self.n_oc*i:self.n_oc*(i+1), self.max_plus_model.np*i:self.max_plus_model.np*(i+1+mu_max)] = H
        self.Ho_p = Ho_p[:, :self.max_plus_model.np*pred_h]

        # add normal constraints for cycles > k, such that they can switch with earlier robots (filling the lower triangular part of the matrix)
        Ho = np.zeros([1, self.max_plus_model.np*mu_max])
        for node, list_of_next_nodes in next_nodes.items():
            for _ in list_of_next_nodes:
                for pred_segment in pred_segments[node]:
                    diff_pred_segments = copy.deepcopy(pred_segments[node])
                    diff_pred_segments.remove(pred_segment)
                    for diff_pred_segment in diff_pred_segments:
                        diff_pred_index = self.max_plus_model.p_indices["p_" + str(diff_pred_segment)]
                        for mu in range(mu_max):
                            constraint = np.zeros([1, self.max_plus_model.np*mu_max])
                            constraint[0, diff_pred_index + (mu_max-mu-1)*self.max_plus_model.np] = beta
                            Ho = np.vstack([Ho, constraint, np.zeros([1, self.max_plus_model.np*mu_max])])
        Ho = Ho[1:]

        for cycle in range(pred_h-1):
            amount = min(mu_max, cycle+1)
            self.Ho_p[self.n_oc*(cycle+1):self.n_oc*(cycle+2), self.max_plus_model.np*(cycle+1-amount):self.max_plus_model.np*(cycle+1)] = Ho[:, -self.max_plus_model.np*amount:]

        # create pointer for ordering constraints with respect to variable p
        pp = np.zeros([1, 1])
        for node, list_of_next_nodes in next_nodes.items():
            for _ in list_of_next_nodes:
                for pred_segment in pred_segments[node]:
                    diff_pred_segments = copy.deepcopy(pred_segments[node])
                    diff_pred_segments.remove(pred_segment)
```

```python
                    for diff_pred_segment in diff_pred_segments:
                        diff_pred_index = self.max_plus_model.p_indices["p_" + str(diff_pred_segment)]
                        for mu in range(mu_max):
                            pp = np.vstack([pp, diff_pred_index+self.max_plus_model.np*mu + 1, 0])
        pp = pp[1:]
        p = pp.shape[0]

        self.o_p_pointer = np.zeros([p*pred_h, 1])
        for cycle in range(mu_max):
            for num in range(int(p/(2*mu_max))):
                self.f_p_pointer[cycle*p + 2*(num*mu_max+cycle):cycle*p + (num+1)*2*mu_max] = pp[num*2*mu_max:(num+1)*2*mu_max-2*cycle]
        return self.Ho_p, self.o_p_pointer

    def o_z(self, pred_h, mu_max):
        H = np.zeros([1, self.max_plus_model.nz])
        next_nodes = {}
        pred_segments = {}
        all_z_indices = {}
        for node in self.graph.multi_in:
            if not isinstance(self.graph.node_dict[node], InputNode):
                row = self.max_plus_model.ordering_matrix[node, :]
                next_nodes[node] = []
                curr_pred_segments = copy.deepcopy(self.graph.multi_in_pred_segments[node])
                pred_segments[node] = curr_pred_segments
                z_index = self.max_plus_model.z_indices["z_" + str(node) + "_" + str(1)]
                all_z_indices[node] = z_index
                for array_of_next_nodes in np.nonzero(row):
                    for next_node in array_of_next_nodes:
                        next_nodes[node].append(next_node)
                        for pred_segment in curr_pred_segments:
                            constraint = np.zeros(self.max_plus_model.nz)
                            diff_pred_segments = copy.deepcopy(curr_pred_segments)
                            diff_pred_segments.remove(pred_segment)
                            for _ in diff_pred_segments:
                                for mu in range(mu_max):
                                    disturbed_constraint = np.zeros(self.max_plus_model.nz)
                                    disturbed_constraint[z_index+mu*len(self.graph.multi_in)] = -beta
                                    H = np.vstack([H, constraint, disturbed_constraint])
        H = H[1:]
        self.Ho_z = np.kron(np.eye(pred_h), H)

        # add normal constraints for cycles > k, such that they can switch with earlier robots (filling the lower triangular part of the matrix)
        Ho = np.zeros([1, self.max_plus_model.nz*mu_max])
        for node, list_of_next_nodes in next_nodes.items():
            z_index = all_z_indices[node]
            for _ in list_of_next_nodes:
                for pred_segment in pred_segments[node]:
                    diff_pred_segments = copy.deepcopy(pred_segments[node])
                    diff_pred_segments.remove(pred_segment)
                    for _ in diff_pred_segments:
                        for mu in range(mu_max):
                            constraint = np.zeros([1, self.max_plus_model.nz*mu_max])
                            constraint[0, z_index + (mu_max-mu)*(self.max_plus_model.nz - len(self.graph.multi_in))] = beta
                            Ho = np.vstack([Ho, constraint, np.zeros([1, self.max_plus_model.nz*mu_max])])
        Ho = Ho[1:]

        for cycle in range(pred_h-1):
            amount = min(mu_max, cycle+1)
            self.Ho_z[self.n_oc*(cycle+1):self.n_oc*(cycle+2), self.max_plus_model.nz*(cycle+1-amount):self.max_plus_model.nz*(cycle+1)] = Ho[:, -self.max_plus_model.nz*amount:]

        # create pointer for ordering constraints with respect to variable z
        pz = np.zeros([1, 1])
        for node, list_of_next_nodes in next_nodes.items():
            z_index = all_z_indices[node]
            for _ in list_of_next_nodes:
                for pred_segment in pred_segments[node]:
                    diff_pred_segments = copy.deepcopy(pred_segments[node])
                    diff_pred_segments.remove(pred_segment)
                    for _ in diff_pred_segments:
                        for mu in range(mu_max):
                            pz = np.vstack([pz, z_index + mu*(self.max_plus_model.nz + len(self.graph.multi_in)) + 1, 0])
        pz = pz[1:]
        p = pz.shape[0]

        self.o_z_pointer = np.zeros([p*pred_h, 1])
        for cycle in range(mu_max):
            add_index = np.zeros([p, 1])
            add_index[::2] = cycle*len(self.graph.multi_in)
            pz = pz + add_index
            for num in range(int(p/(2*mu_max))):
                self.o_z_pointer[cycle*p + 2*(num*mu_max+cycle):cycle*p + (num+1)*2*mu_max] = pz[num*2*mu_max:(num+1)*2*mu_max-2*cycle]
        return self.Ho_z, self.o_z_pointer

    def o_scalar(self, pred_h):
        scalar = np.zeros([self.n_oc, 1])
        scalar[1::2] = -beta
        self.o_scalar_vector = np.kron(np.ones([pred_h, 1]), scalar)
        return self.o_scalar_vector
```

Appendix B

Optimization

B-1 The optimization problem

```
import numpy as np
import gurobipy as gp

from gurobipy import GRB
from class_constraints import Constraints, assemble_vectors


class Optimization:

    def __init__(self, graph, max_plus_model, constraints, pred_h,
       gamma_max, robots):

        self.constraints = constraints
        self.max_plus_model = max_plus_model
        self.graph = graph
        self.pickup_time = 2

        try:

            """ Create a new model """
            m = gp.Model("test")

            """ Set objective """
            m.modelSense = GRB.MINIMIZE

            """ Create variables """
            # Objective
            max_x = m.addMVar(shape=1, obj=1, vtype=GRB.CONTINUOUS, name=
                "max_x")

            # Routing
            x = m.addMVar(shape=max_plus_model.nx*pred_h, obj=0.01, vtype=GRB.CONTINUOUS, name="x")
            s_ij = m.addMVar(shape=max_plus_model.ns*pred_h, vtype=GRB.BINARY, name="s_ij")
            w_l = m.addMVar(shape=max_plus_model.nw*pred_h, vtype=GRB.BINARY, name="w_l")

            # Coupling
            c_k_gamma = m.addMVar(shape=max_plus_model.nc*(pred_h-robots), vtype=GRB.BINARY, name="c_k_gamma")

            # Ordering
            p_l = m.addMVar(shape=max_plus_model.np*pred_h, vtype=GRB.BINARY, name="p_l")
            f_l_mu = m.addMVar(shape=max_plus_model.nf*pred_h, vtype=GRB.BINARY, name="f_l_mu")
            z_i_mu = m.addMVar(shape=max_plus_model.nz*pred_h, vtype=GRB.BINARY, name="z_i_mu")

            # Synchronization
            b_i_mu = m.addMVar(shape=max_plus_model.nb*pred_h, vtype=GRB.BINARY, name="b_i_mu")

            """ Add constraints """
            # Routing
            self.distortion_routing_rhs = assemble_vectors(self.constraints.routing_pointer, [], [self.constraints.total_theta])
            m.addConstr(self.constraints.routing_x @ x + self.constraints.routing_s @ s_ij <= self.distortion_routing_rhs, name="routing")

            # Input
            m.addConstr(self.constraints.input_x @ x + self.constraints.input_s @ s_ij <= self.constraints.input_rhs, name="input")
            m.addConstr(self.constraints.input_add_s @ s_ij == self.constraints.input_add_rhs, name="additional_input_path")

            # Coupling
            m.addConstr(self.constraints.coupling_x @ x + self.constraints.coupling_s @ s_ij + self.constraints.coupling_c @ c_k_gamma <= self.constraints.coupling_rhs, name="coupling")
            m.addConstr(self.constraints.last_node_x @ x + self.constraints.last_node_s @ s_ij + self.constraints.last_node_c @ c_k_gamma <= self.constraints.last_node_rhs, name="last_node")

            # Ordering
            m.addConstr(self.constraints.ordering_f_x @ x + self.constraints.ordering_f_p @ p_l + self.constraints.ordering_f_f @ f_l_mu <= self.constraints.ordering_f_rhs, name="following")
            m.addConstr(self.constraints.ordering_e_x @ x + self.constraints.ordering_e_f @ f_l_mu <= self.constraints.ordering_e_rhs, name="splitting")
            m.addConstr(self.constraints.ordering_o_x @ x + self.constraints.ordering_o_p @ p_l + self.constraints.ordering_o_z @ z_i_mu <= self.constraints.ordering_o_rhs, name="ordering")

            # Synchronization
            m.addConstr(self.constraints.synchr_x @ x + self.constraints.synchr_s @ s_ij + self.constraints.synchr_b @ b_i_mu <= self.constraints.synchr_rhs, name="synchronization")

            # Additional constraints
            m.addConstr(self.constraints.add_s_ij_s @ s_ij + self.constraints.add_s_ij_w @ w_l == self.constraints.add_s_ij_rhs, name="equality_s_ij")
            m.addConstr(self.constraints.add_w_l_w @ w_l == self.constraints.add_w_l_rhs, name="equality_w_l")
            m.addConstr(self.constraints.add_p_l_p @ p_l + self.constraints.add_p_l_w @ w_l == self.constraints.add_p_l_rhs, name="equality_p_l")
            m.addConstr(self.constraints.add_f_l_mu_f @ f_l_mu + self.constraints.add_f_l_mu_z @ z_i_mu == self.constraints.add_f_l_mu_rhs, name="equality_f_l_mu")
            m.addConstr(self.constraints.add_x_x @ x + self.constraints.add_x_w @ w_l <= self.constraints.add_x_rhs, name="last_visited")
            m.addConstr(self.constraints.add_to_input_s @ s_ij + self.constraints.add_to_input_w @ w_l == self.constraints.add_to_input_rhs, name="to_input")
            m.addConstr(self.constraints.add_from_input_s @ s_ij + self.constraints.add_from_input_c @ c_k_gamma <= self.constraints.add_from_input_rhs, name="from_input")
            m.addConstr(self.constraints.add_c_coupling_horizontal @ c_k_gamma == self.constraints.add_coupling_horizontal_rhs, name="coupling_horizontal")
            m.addConstr(self.constraints.add_c_coupling_vertical_ineq @ c_k_gamma <= self.constraints.add_coupling_vertical_ineq_rhs, name="coupling_vertical_ineq")
            m.addConstr(self.constraints.add_c_coupling_vertical @ c_k_gamma == self.constraints.add_coupling_vertical_rhs, name="coupling_vertical")
            m.addConstr(self.constraints.target_w @ w_l == self.constraints.target_rhs, name="targets")

            # Objective that includes event times of all parcels
            for i in range(pred_h*max_plus_model.nx):
```

```python
            m.addConstr(x[i] <= max_x, name="minimize_max_x_value[" + str(i) + "]")

        # Add penalty to states of inputs that should not be visited
        self.q_list = []
        for k in range(pred_h):
            self.q_list.append(self.constraints.input_info[k]["input"])

        for num, k in enumerate(self.q_list):
            successor_list = graph.input_succ[k]
            for successor in successor_list:
                s_index = max_plus_model.s_indices["s_" + str(k) + "_" + str(successor)]
                s_ij[s_index + num*max_plus_model.ns].Obj = 100

        """ Save model """
        m.write('model_routing.lp')

        """ Optimize model """
        m.optimize()

        # Extract states and binary variables for each node for each cycle looked ahead
        self.states = np.zeros([pred_h, max_plus_model.nx])
        self.arcs = np.zeros([pred_h, max_plus_model.ns])
        self.routes = np.zeros([pred_h, max_plus_model.nw])
        self.segments = np.zeros([pred_h, max_plus_model.np])
        self.follow = np.zeros([pred_h, max_plus_model.nf])
        self.order = np.zeros([pred_h, max_plus_model.nz])
        self.synchronizations = np.zeros([pred_h, max_plus_model.nb])
        self.coupled = np.zeros([pred_h-robots, max_plus_model.nc])
        self.targets = np.zeros([pred_h, 1])
        for i in range(pred_h):
            self.states[i, :] = x.X[i*max_plus_model.nx:(i+1)*max_plus_model.nx]
            self.arcs[i, :] = s_ij.X[i*max_plus_model.ns:(i+1)*max_plus_model.ns]
            self.routes[i, :] = w_l.X[i*max_plus_model.nw:(i+1)*max_plus_model.nw]
            self.segments[i, :] = p_l.X[i*max_plus_model.np:(i+1)*max_plus_model.np]
            self.follow[i, :] = f_l_mu.X[i*max_plus_model.nf:(i+1)*max_plus_model.nf]
            self.order[i, :] = z_i_mu.X[i*max_plus_model.nz:(i+1)*max_plus_model.nz]
            self.synchronizations[i, :] = b_i_mu.X[i*max_plus_model.nb:(i+1)*max_plus_model.nb]
            if i >= robots:
                self.coupled[i-robots, :] = c_k_gamma.X[(i-robots)*max_plus_model.nc:(i+1-robots)*max_plus_model.nc]
                if i < gamma_max:
                    self.coupled[i-robots, -(gamma_max-i):] = 0.5
            self.targets[i] = self.constraints.targets[i]

        self.visited_states = np.zeros([pred_h, max_plus_model.nx])
        for k in range(pred_h):
            path_number = np.where(self.routes[k] == 0)[0][0]
            path = graph.paths[path_number]
            for node in path.all_nodes:
                self.visited_states[k, node] = self.states[k, node]

    except gp.GurobiError as e:
        print('Error code ' + str(e.errno) + ': ' + str(e))

    except AttributeError:
        print('Encountered an attribute error')
```

B-2 A setup for MPS

The code below is the beginning of the implementation of model predictive scheduling (MPS). Every 5 seconds (`step`), the model gets updated with the function `run_mpc` which calls method `update`. The update does not measure real-time data on the robots yet, but only uses the locations of the robots as predicted in the optimization. Lists of active and finished robots are constructed and updated. The method `adapt_double_mu` is the beginning of a method that adapts the constraint and index matrices for ordering and synchronization constraints according to the structure of the active robots list. This method is not finished yet. Method `adapt_single_mu` should adapt the constraint and index matrices for coupling constraints when the active robots list has a structural change.

```
import copy
import operator

import numpy as np
import sched
import time

from main import equinox, mp_model, Np, mu, r, equinox_constraints, equinox_opt
from class_optimization import Optimization
from class_constraints import Constraints


class MPC:

    def __init__(self, graph, max_plus_model, pred_h, mu_max, robots, tot_parcels):
        self.max_plus_model = max_plus_model
        self.graph = graph
        self.active_robots = [k for k in range(robots)]
        self.horizon = [k for k in range(pred_h)]
        self.finished_robots = []
        self.past_robots = []
```

```python
        self.n_fut = np.zeros([Np, 1])      # amount of robots > k' that finished before k' (for k' = k,...,k+Np-1)
        self.n_past = np.zeros([Np, 1])     # amount of robots k: k'-mu_max < k < k' that finished already
        self.threshold = np.zeros([Np, 1])  # highest number allowed in pointers (in case that a robot > k finishes before k)

        # self.constraints = Constraints(graph, max_plus_model, pred_h, mu_max, robots)
        self.init_constraints = equinox_constraints
        self.constraints = copy.deepcopy(self.init_constraints)
        # self.optimization = Optimization(graph, max_plus_model, self.constraints, pred_h)
        self.init_optimization = equinox_opt
        self.optimization = None
        self.test = []
        self.changed = []

        finish_prediction = {}
        for robot in self.horizon:
            finish_prediction[robot] = round(max(self.init_optimization.visited_states[robot, :]), 1)
        self.sorted_finish_prediction = dict(sorted(finish_prediction.items(), key=operator.itemgetter(1)))

    def update(self, curr_t):
        """ Call this function each time step, include changes in structure when a robot finishes a job,
        update constraints with new measured information and run optimization with updated constraints """
        sort_finished_robots = {}
        for robot in self.active_robots:
            if max(self.init_optimization.visited_states[robot, :]) < curr_t:
                sort_finished_robots[robot] = max(self.init_optimization.visited_states[robot, :])
        sorted_finished_robots = dict(sorted(sort_finished_robots.items(), key=operator.itemgetter(1)))
        if max(self.init_optimization.visited_states[self.active_robots[0], :]) < curr_t:
            self.shift_horizon(len(sorted_finished_robots), curr_t)
        for _ in range(len(sorted_finished_robots)):
            if max(self.active_robots) < Np-1:
                self.active_robots.append(max(self.active_robots) + 1)
        for robot in sorted_finished_robots.keys():
            self.finished_robots.append(robot)
            self.active_robots.remove(robot)
        self.finished_robots = list(dict.fromkeys(self.finished_robots))

        active_pred = copy.deepcopy(self.active_robots)
        while len(active_pred) < Np and active_pred:
            active_pred.append(max(active_pred) + 1)
        self.n_fut = np.zeros([Np, 1])
        self.n_past = np.zeros([Np, 1])
        for count, robot in enumerate(active_pred):
            max_num = min(len(active_pred) - 1, count+mu)
            difference = active_pred[max_num] - robot - min(max_num-count, mu)
            self.n_fut[count] = max(0, difference)
            max_back = max(0, count-mu)
            difference_back = robot - active_pred[max_back] - min(count, mu)
            self.n_past[count] = max(0, difference_back)
            self.threshold[count] = max(0, mu-(robot-active_pred[0]))
        if np.any(self.n_fut != 0) or np.any(self.n_past != 0):     # and if any robot finished! Keep track of this somehow
            self.constraints.ordering_o_x = self.adapt_double_mu(self.init_constraints.ordering_o_x, self.init_constraints.ordering_o_x_pointer, self.init_constraints.noc, self.max_plus_model.nx)
        self.test = self.init_constraints.ordering_o_x - self.constraints.ordering_o_x
        return

    def shift_horizon(self, n, t):
        """ Call this function when k finishes and the horizon shifts """
        shift_amount = 0
        for robot in range(n):
            if max(self.init_optimization.visited_states[self.active_robots[robot], :]) < t:
                shift_amount += 1
        print("shift with " + str(shift_amount))
        return

    def adapt_double_mu(self, matrix, pointer, n_con, n_var):
        ad_matrix = copy.deepcopy(matrix)
        ad_pointer = copy.deepcopy(pointer)
        change_odd_rows = {}
        change_even_rows = {}
        for i in range(Np-1):
            change_odd_rows[i] = {"row": [], "pointer": []}
            change_even_rows[i] = {"row": [], "pointer": []}
            for row in range(int(n_con/2)):
                for m in range(self.n_fut[i]):
                    try:
                        if (ad_matrix[i*n_con + 2*row+1, (i+mu-m)*n_var:(i+1+mu-m)*n_var] != 0).any():
                            change_odd_rows[i]["row"].append(2*row+1)
                            var_ind = np.where(ad_matrix[i*n_con + 2*row+1, (i+mu-m)*n_var:(i+1+mu-m)*n_var] != 0)[0][0]
                            change_odd_rows[i]["pointer"].append((m+mu)*n_var + var_ind + 1)
                            ad_matrix[i*n_con + 2*row+1, (i+mu-m)*n_var:(i+1+mu-m)*n_var] = np.zeros([1, n_var])
                    except IndexError:
                        pass
                for m in range(self.n_past[i]):
                    try:
                        if (ad_matrix[i*n_con + 2*row, (i-mu+m)*n_var:(i+1-mu+m)*n_var] != 0).any():
                            change_even_rows[i]["row"].append(2*row)
                            var_ind = np.where(ad_matrix[i*n_con + 2*row, (i-mu+m)*n_var:(i+1-mu+m)*n_var] != 0)[0][0]
                            change_even_rows[i]["pointer"].append((m+mu)*n_var + var_ind + 1)
                            ad_matrix[i*n_con + 2*row, (i-mu+m)*n_var:(i+1-mu+m)*n_var] = np.zeros([1, n_var])
                    except IndexError:
                        pass
                if ad_pointer[i*n_con + 2*row] > self.threshold[i]*n_var:
                    pass
        #         adapt pointer and matrix here
        return adapted

    def adapt_single_mu(self):
        return

Ntot = 20
equinox_mpc = MPC(equinox, mp_model, Np, mu, r, Ntot)

s = sched.scheduler(time.time, time.sleep)
step = 5
t0 = time.time()
t_end = 100
counter = 0


def run_mpc(schedule):
    global counter
    t = round(time.time()-t0, 1)
    print()
    print("run " + str(counter) + ": t = " + str(t))
    equinox_mpc.update(t)
    print("finished = " + str(equinox_mpc.finished_robots))
    print("active = " + str(equinox_mpc.active_robots))
    if t < t_end-step:
        s.enter(step, 1, run_mpc, (schedule,))
        counter += 1

s.enter(step, 1, run_mpc, (s,))
s.run()
```

Appendix C

Visualization

```python
import numpy as np
import networkx as nx

import math
import copy

from main import equinox, equinox_opt, mp_model, Np, r
from matplotlib import pyplot as plt
from matplotlib import animation
from class_node import InputNode


def round_decimals_up(number: float, decimals: int = 2):
    if not isinstance(decimals, int):
        raise TypeError("decimal places must be an integer")
    elif decimals < 0:
        raise ValueError("decimal places has to be 0 or more")
    elif decimals == 0:
        return math.ceil(number)

    factor = 10 ** decimals
    return math.ceil(number * factor) / factor


class Visualization:

    def __init__(self, graph, max_plus_model, optimization, pred_h,
            robots, figure):
        self.active = [k for k in range(robots)]
        self.robots = [k for k in range(optimization.states.shape[0])]
        self.node_coordinates = graph.node_positions
        self.visited_nodes = optimization.visited_states
        self.sorted_entry_times = {k: {"nodes": [], "times": [], "coordinates": [], "arc_lengths": [], "control_points": [], "input_time": 0, "target": None} for k in self.robots}
        for k in self.robots:
            path_number = np.where(optimization.routes[k] == 0)[0][0]
            path = graph.paths[path_number]
            self.sorted_entry_times[k]["nodes"] = copy.deepcopy(path.all_nodes)
            last_node = self.sorted_entry_times[k]["nodes"][-1]
            back_to_input_ids = graph.node_dict[last_node].connects_to
            back_to_input_node = -1
            for node_id in back_to_input_ids:
                node = graph.find_node_number(node_id)
                if isinstance(graph.node_dict[node], InputNode):
                    back_to_input_node = node
                    break
            self.sorted_entry_times[k]["nodes"].append(back_to_input_node)
            self.sorted_entry_times[k]["input_time"] = optimization.constraints.input_times[k]
            self.sorted_entry_times[k]["target"] = optimization.constraints.targets[k]
            for node in self.sorted_entry_times[k]["nodes"]:
                time = round_decimals_up(self.visited_nodes[k, node], 1)
                self.sorted_entry_times[k]["times"].append(time)
                coordinate = self.node_coordinates[node]
                self.sorted_entry_times[k]["coordinates"].append(coordinate)
            for num, node in enumerate(self.sorted_entry_times[k]["nodes"][1:]):
                edge = (path.all_nodes[num], node)
                weight = graph.edge_weight_dict[edge]
                self.sorted_entry_times[k]["arc_lengths"].append(weight)
                control_points = graph.edge_dict[edge].control_points
                self.sorted_entry_times[k]["control_points"].append(control_points)

        self.couples = {}
        for k in self.robots:
            coupled_to = None
            for num in range(pred_h-robots):
                try:
                    if abs(optimization.coupled[num, num + robots-1 - k]) == 0 and num+robots-1-k >= 0:
                        coupled_to = num + robots
                        break
                except IndexError:
                    pass
            if coupled_to:
                new_time = round_decimals_up(self.visited_nodes[coupled_to, self.sorted_entry_times[coupled_to]["nodes"][0]], 1)
            else:
                new_time = round_decimals_up(self.sorted_entry_times[k]["times"][-2] + self.sorted_entry_times[k]["arc_lengths"][-1], 1) + 2
            self.sorted_entry_times[k]["times"][-1] = new_time
            self.couples[k] = coupled_to

        for input_node, pred_list in graph.input_pred.items():
            for node in pred_list:
                incoming_order = np.argsort(self.visited_nodes[:, node])
                for robot in incoming_order:
                    if self.visited_nodes[robot, node] == 0:
                        incoming_order = np.delete(incoming_order, np.where(incoming_order == robot)[0])
                for num, robot in enumerate(incoming_order[:-1]):
                    if self.sorted_entry_times[incoming_order[num+1]]["times"][-2] - self.sorted_entry_times[robot]["times"][-1] < 0:
                        self.sorted_entry_times[incoming_order[num+1]]["times"][-2] = self.sorted_entry_times[robot]["times"][-1]

        self.dt = 0.05
        self.T = 0
        for k in self.robots:
            max_t = max(self.sorted_entry_times[k]["times"])
            if max_t > self.T:
                self.T = max_t + 1
        self.T = math.ceil(self.T/self.dt) + 1
        self.x = np.zeros([self.T, len(self.robots)])
        self.y = np.zeros([self.T, len(self.robots)])
        for k in self.robots:
            t = 0
            row = 0
            self.x[row, k] = self.sorted_entry_times[k]["coordinates"][0][0]
            self.y[row, k] = self.sorted_entry_times[k]["coordinates"][0][1]
            reference = max(self.sorted_entry_times[k]["input_time"][0], self.sorted_entry_times[k]["times"][0])
            step = round(self.dt*self.sorted_entry_times[k]["arc_lengths"][0]/(self.sorted_entry_times[k]["times"][1]-reference), 4)
            for num, i in enumerate(self.sorted_entry_times[k]["times"][:-1]):
                xy_node = self.sorted_entry_times[k]["coordinates"][num]
                xy_node_next = self.sorted_entry_times[k]["coordinates"][num+1]
                direction = (xy_node_next[0] - xy_node[0], xy_node_next[1] - xy_node[1])
                normalized_dir = direction/np.linalg.norm(direction)
                control_points = copy.deepcopy(self.sorted_entry_times[k]["control_points"][num])
                xy_next = xy_node
                t += self.dt
                row += 1
                while self.sorted_entry_times[k]["input_time"][0]-t >= self.dt/2 or self.sorted_entry_times[k]["times"][0]-t >= self.dt/2:
                    self.x[row, k], self.y[row, k] = xy_next
                    t += self.dt
                    row += 1
                while self.sorted_entry_times[k]["times"][num+1]-t >= self.dt/2:
                    next_node = self.sorted_entry_times[k]["nodes"][num+1]
                    curr_node = self.sorted_entry_times[k]["nodes"][num]
                    if next_node in graph.multi_in or curr_node in graph.block_number_dict.keys():
                        nom_time = max_plus_model.routing_matrix[next_node, curr_node]
                        if self.sorted_entry_times[k]["times"][num+1] - i > nom_time:
                            step = round(self.dt*self.sorted_entry_times[k]["arc_lengths"][num]/nom_time, 4)
                            while self.sorted_entry_times[k]["times"][num+1] - t > nom_time:
                                self.x[row, k], self.y[row, k] = xy_node
                                t += self.dt
                                row += 1
                    elif curr_node == self.sorted_entry_times[k]["target"]:
                        step = round(self.dt*self.sorted_entry_times[k]["arc_lengths"][num]/(self.sorted_entry_times[k]["times"][num+1]-i-2), 4)
                        while i - t >= -2:
                            self.x[row, k], self.y[row, k] = xy_next
                            t += self.dt
                            row += 1
                    elif curr_node in graph.input_pred.keys():
                        step = round(self.dt*self.sorted_entry_times[k]["arc_lengths"][num]/(self.sorted_entry_times[k]["times"][num+1]-reference-2), 4)
                        while reference - t >= -2:
                            self.x[row, k], self.y[row, k] = xy_next
                            t += self.dt
                            row += 1
                    elif next_node in graph.input_pred.keys():
                        step = round(self.dt*self.sorted_entry_times[k]["arc_lengths"][num]/(self.sorted_entry_times[k]["times"][num+1]-i), 4)
                    xy_cp = xy_node
                    while control_points:
                        xy_cp_next = control_points[0][:2]
                        direction = (xy_cp_next[0] - xy_cp[0], xy_cp_next[1] - xy_cp[1])
                        normalized_dir = direction/np.linalg.norm(direction)
                        xy_next += normalized_dir*step
                        self.x[row, k], self.y[row, k] = xy_next
                        t += self.dt
                        row += 1
                        if abs(xy_next[0]-xy_cp_next[0]) < step*0.9 and abs(xy_next[1]-xy_cp_next[1]) < step*0.9:
                            control_points.pop(0)
                            xy_cp = xy_cp_next
                            direction = (xy_node_next[0] - xy_cp[0], xy_node_next[1] - xy_cp[1])
                            normalized_dir = direction/np.linalg.norm(direction)
                    xy_next += normalized_dir*step
                    self.x[row, k], self.y[row, k] = xy_next
                    t += self.dt
                    row += 1
                if self.sorted_entry_times[k]["times"][-1]-t >= self.dt/2:
                    self.x[row, k], self.y[row, k] = xy_node_next
                    step = round(self.dt*self.sorted_entry_times[k]["arc_lengths"][num+1]/(self.sorted_entry_times[k]["times"][num+2]-self.sorted_entry_times[k]["times"][num+1]), 4)
                elif self.sorted_entry_times[k]["times"][-1]-t < self.dt/2:
                    self.x[row:, k], self.y[row:, k] = xy_node_next

        ax = plt.gca()
        self.circles = []
        for k in self.robots:
            self.circles.append(plt.Circle((self.x[0, k], self.y[0, k]), 0.15, color='black', fc='r', zorder=3))
        ax.set_aspect('equal')
        for num, c in enumerate(self.circles):
            ax.add_patch(c)


def animate(time):
    for k in visual_graph.active:
        if time*visual_graph.dt > max(visual_graph.sorted_entry_times[k]["times"]):
            visual_graph.active.remove(k)
            if visual_graph.couples[k]:
                visual_graph.active.append(visual_graph.couples[k])
    for t, circle in enumerate(visual_graph.circles):
        if t in visual_graph.active:
            circle.set_visible(True)
            circle.center = (visual_graph.x[time, t], visual_graph.y[time, t])
            if circle.center == visual_graph.node_coordinates[visual_graph.sorted_entry_times[t]["target"]]:
```

```python
                circle.set_facecolor('g')
            if circle.center == visual_graph.sorted_entry_times[t]["coordinates"][0]:
                circle.set_facecolor('r')
        else:
            circle.set_visible(False)
    return visual_graph.circles

""" Draw graph """
fig = plt.figure(figsize=(20, 5))
nx.draw_networkx_edge_labels(equinox.to_draw, equinox.draw_node_positions, edge_labels=equinox.draw_edge_labels)
nx.draw(equinox.to_draw, equinox.draw_node_positions, node_size=equinox.draw_sizes, labels=equinox.draw_labels, node_color=equinox.draw_color_map, nodelist=sorted(equinox.to_draw.nodes()), with_labels=True)
visual_graph = Visualization(equinox, mp_model, equinox_opt, Np, r, fig)

""" Animate """
anim = animation.FuncAnimation(fig, animate, frames=visual_graph.T, interval=1000*visual_graph.dt, repeat=False, blit=True)
```

Bibliography

[1] M. Silva, “On the history of Discrete Event Systems,” *Annual Reviews in Control*, vol. 45, pp. 213–222, 2018.

[2] M. Alirezaei, T. van den Boom, and R. Babuska, “Max-plus algebra for optimal scheduling of multiple sheets in a printer,” in *Proceedings of the American Control Conference*, (Montréal, Canada), pp. 1973–1978, 2012.

[3] B. Kersbergen, *Modeling and Control of Switching Max-Plus-Linear Systems*. PhD thesis, Delft University of Technology, 2015.

[4] B. Kersbergen, G.A.D. Lopes, T. van den Boom, B. de Schutter, and R. Babuška, “Optimal gait switching for legged locomotion,” in *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS’11)*, (San Francisco, California), pp. 2729–2734, 2011.

[5] T. van den Boom, M. van den Muijsenberg, and B. de Schutter, “Model predictive scheduling of semi-cyclic discrete-event systems using switching max-plus linear models and dynamic graphs,” *Discrete Event Dynamic Systems: Theory and Applications*, vol. 30, no. 4, pp. 635–669, 2020.

[6] M. van den Muijsenberg, “Scheduling using max-plus algebra,” Master’s thesis, Delft University of Technology, 2015.

[7] Prime Vision, “Company Profile.” https://www.primevision.com/company-profile/, 2020. Accessed 15.04.2021.

[8] Prime Vision, “Autonomous Sorting.” https://www.primevision.com/autonomous-sorting/, 2020. Accessed 15.04.2021.

[9] M. Drótos, P. Györgyi, M. Horváth, and T. Kis, “Suboptimal and conflict-free control of a fleet of AGVs to serve online requests,” *Computers and Industrial Engineering*, vol. 152, 2021.

[10] G. van Rossum and F.L. Drake, *Python 3 Reference Manual.* Scotts Valley, CA: CreateSpace, 2009.

[11] Prime Vision: internal report (confidential), “Machine overview - System introduction.” Internal web page, 2020. Accessed 12.03.2021.

[12] B. Heidergott, G.J. Olsder, and J. van der Woude, *Max Plus at Work. Modeling and analysis of synchronized systems: A course on max-plus algebra and its applications*, vol. 13. Princeton, NJ: Princeton University Press, 2005.

[13] B. de Schutter and T. van den Boom, “Max-plus algebra and max-plus linear discrete event systems: An introduction,” in *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES’08)*, (Göteborg, Sweden), pp. 36–42, 2008.

[14] Gurobi Optimization, LLC, “Gurobi Optimizer Reference Manual.” https://www.gurobi.com/documentation/9.5/refman/index.html, 2021. Accessed 23.08.2021.

[15] T. van den Boom and B. de Schutter, “Modeling and control of switching max-plus-linear systems with random and deterministic switching,” *Discrete Event Dynamic Systems: Theory and Applications*, vol. 22, no. 3, pp. 293–332, 2012.

[16] T. van den Boom and B. de Schutter, “Model predictive control for perturbed max-plus-linear systems: A stochastic approach,” *International Journal of Control*, vol. 77, no. 3, pp. 302–309, 2004.

[17] A. Schrijver, *Theory of Linear and Integer Programming.* Chichester: Wiley, 1986.

[18] A. Atamtürk and M.W.P. Savelsbergh, “Integer-programming software systems,” *Annals of Operations Research*, vol. 140, no. 1, pp. 67–124, 2005.

[19] R.W.A. Neijenhuis, “Modelling and Analysis Using a Switching Stochastic Max-Plus Linear Model,” Master’s thesis, Delft University of Technology, 2017.

[20] B. de Jong, “Throughput and Stabilisability Analysis of Mode-Constrained Stochastic Switching Max-Plus Linear Systems,” Master’s thesis, Delft University of Technology, 2022.

[21] R. Tempo, G. Calafiore, and F. Dabbene, *Randomized algorithms for analysis and control of uncertain systems: with applications.* Springer, second ed., 2013.

[22] N. Pisaruk, “Mixed Integer Programming Class Library (MIPCL),” in *Tanaev’s Readings Proceedings of the 7th International Conference*, (Minsk, Belarus), 2016.

Glossary

List of Symbols

$[A]_{i,j}$	Element (i,j) of matrix A
$\bar{w}$	The adjoint of max-plus binary value w
$\gamma_{\max}$	Maximum amount of previous robots that can turn into a new robot at an input node
$\mathbb{B}$	The set of binary numbers $\{0,1\}$
$\mathbb{B}_\varepsilon$	The set of max-plus binary numbers $\{0,\varepsilon\}$
$\mathbb{R}_\varepsilon$	The set of real numbers including $-\infty$
$\mathbb{R}$	The set of real numbers
$\mathbb{R}^{m\times n}$	An $m\times n$ matrix with real numbers
$\mathcal{D}(A)$	Set of edges of graph $\mathcal{G}(A)$
$\mathcal{E}_{m\times n}$	An $m\times n$ matrix containing only ε
$\mathcal{G}(A)$	Precedence graph of matrix A
$\mathcal{N}(A)$	Set of nodes of graph $\mathcal{G}(A)$
$\mathcal{N}_{\text{ord}}$	Set of nodes with multiple incoming edges
$\mu_{\max}$	Maximum amount of robots ahead and behind taken into account
$\odot$	Max-plus Schur product
$\oplus$	Max-plus addition (maximization)
$\otimes$	Max-plus multiplication (addition)
$\pi(i)$	The set of predecessor nodes of node i
$\pi_s(i)$	The set of segments that end in node i
$\sigma(i)$	The set of successor nodes of node i
$\tau_{i,j}(k)$	Travel time for robot k from node i to node j
$\theta(k)$	Vector with travel times of all robots
$\tilde{u}_e(k)$	Future input values
$\tilde{x}(k)$	Vector of estimated future states
ε	The max-plus zero element $(-\infty)$
$\zeta(k)$	Optimization vector

A^*	Kleene star of matrix A
A^{ord}	Ordering matrix for an SMPL system
A^{route}	Routing matrix for an SMPL system
A^{syn}	Synchronization matrix for an SMPL system
$b(i)$	The set nodes that are too close to a successor node of node i
$b_0(k)$	Vector containing known decision variables
$b_{i,\mu}(k-\mu)$	Binary decision variable that determines the synchronization order between robots k and $k-\mu$ in node i
$c_k(k+\gamma)$	Binary decision variable that determines the coupling between robots k and $k+\gamma$
E_n	The max-plus algebraic identity matrix of size $n \times n$ with 0 on the diagonal and ε elsewhere
F	Index matrix
F_μ^*	Index matrix pointing μ robots ahead and to robots behind that finished earlier
F_μ	Index matrix pointing μ robots ahead
$f_{l,\mu}(k-\mu)$	Binary decision variable that determines the order between robots k and $k-\mu$ on segment l
H	Constraint matrix in the optimization problem
H_μ^+	Part of the constraint matrix that refers to μ robots behind
H_μ	Part of the constraint matrix that refers to μ robots ahead
$J(k)$	Objective function
$J_u(k)$	Objective function regarding the input nodes
$J_x(k)$	Objective function regarding the state times
k	Product (robot) counter
L_{syn}	Set of synchronization modes
n	Number of states (nodes)
N_p	Prediction horizon
$p_l(k)$	Binary decision variable that determines if segment l is traveled by robot k
r	Amount of active robots
$s_{i,j}(k)$	Binary decision variable that determines if edge (i,j) is used by robot k
$u_e(k)$	The time instant that the parcel for robot k is scanned and ready for pickup
$v(k)$	Control vector containing all binary decision variables
$w^\flat$	The max-plus binary variable w converted to a conventional binary variable
$w_\ell(k)$	Binary decision variable that determines if route ℓ is traveled by robot k
$x(k)$	State vector: time instants that internal events happen for robot k
$z_{i,\mu}(k-\mu)$	Binary decision variable that determines the order between robots k and $k-\mu$ in node i

List of Acronyms

DES	discrete event system
MILP	mixed integer linear programming
MPC	model predictive control
MPL	max-plus linear
MPS	model predictive scheduling
SMPL	switching max-plus linear