Institute of Automation and Information Systems
TUM School of Engineering and Design
Technical University of Munich

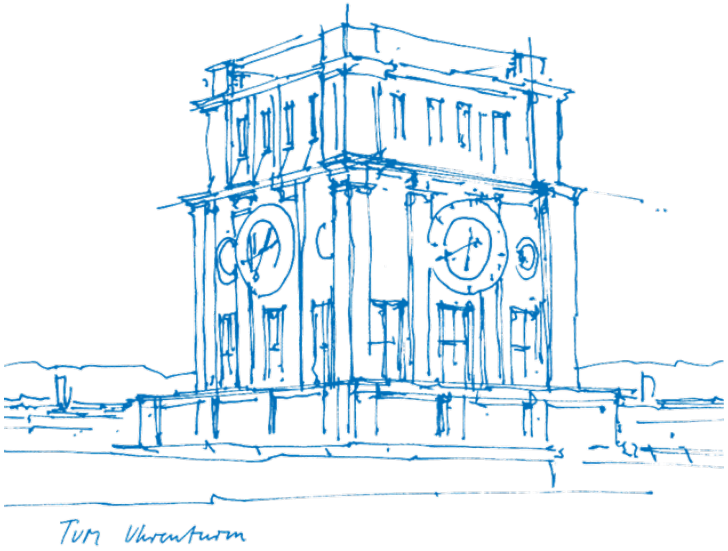
Master's Thesis in Robotics, Cognition, Intelligence

Planning Magnetic Levitation Operating Strategies for PLC-Controlled Intralogistics

Version Adapted as Tech Report for the Publication

Interaction-Minimizing Roadmap Optimization for High-Density Multi-Agent Path Finding

Sören Arne Weindel



Abstract

Modern industry increasingly demands customizability from each element of their workflow and factories. A prominent example for this is the advent of smart Automated Guided Vehicles (AGV) in intralogistics tasks, which are able to autonomously navigate fields in an effort to optimally deliver products. One such application, and the central concern of this work, is the application of fleets of magnetically propelled movers to transport, collect, sort and group products between multiple stations of a larger factory assembly. Traditional approaches to this task make use of inflexible conveyor belts or pick-and-place machines, limited in their ability of to efficiently adapt to new requirements. Most state-of-the-art solutions for controlling fleets of AGVs rely on smart, distributed systems, which are able to independently make real-time routing decisions. However, most factories rely on centralized Programmable Logic Controllers (PLC), designed around proven technologies and bound to industry standards such as IEC 61131. This makes the adaptation and integration of modular approaches and consequently their customizability slow and complex.

This work presents an AGV control approach capable of offloading large parts of the computational expense into an offline preprocessing step, making use of an iterative force gradient optimization to lay out a graph of unidirectional lanes and a novel item sequence approach to efficiently ensure adherence to a specified product arrival order at the offloading points. Furthermore, a concept for a PLC-based, reusable and efficient real-time controller and an accompanying simulator are presented, used to verify the fitness of the generated solutions.

Contents

1 Introduction

With modern industry increasingly being expected to customize their production to the specific needs of each customer [1], the ability to efficiently adapt existing workflows and facilities to new requirements is a key factor for manufacturers to stay competitive [2, 3]. One area of interest in this change is the restructuring and modularization of production lines [4], which involves innovative approaches to the transportation of products between individual stations [5], referred to as intralogistics. Traditionally, such solutions are accomplished using conveyor belts and pick-and-place machines for transporting, sorting and grouping tasks [6]. While these systems are typically designed to be modular and are laid out with reconfigurations in mind, their nature nevertheless limits the flexibility of the system and therefore the ability to dynamically adapt to changing requirements of the final product composition. Most changes to their configuration require manual operator intervention modifying the hardware, which makes frequent adjustments economically infeasible.

An alternative to the more traditional approaches, which has become more prominent over the last years [7], are Automated Guided Vehicles (AGV). These mobile robots are usually employed in fleets, operating along predefined paths or inside dedicated areas and move products in coordination with each other and their attached subassemblies. While fully autonomous AGV systems are commonplace today, their integration into existing assemblies and workflows remains challenging [8]. Despite this, they are a viable alternative, as they can be easily adapted to a variety of arrangements within a facility. In addition to transporting tasks, AGVs are able to sort and group products without the need for any additional hardware. In this, they are well-suited for managing the product flow between multiple preceeding and following subassemblies, allowing products to be dynamically mixed and distributed to where they are required.

An upcoming technology in small-scale AGVs relies on linear magnetic propulsion instead of wheel- or rail-based approaches. This allows for sealed, solid state systems operating without any physical contact between movers and the environment. Additionally, they have the ability to move in six degrees of freedom as shown in Figure 1.1, allowing for the efficient and highly customizable transport of products. A number of systems relying on this technology have been released on the market in the past years, such as the Planar Motor XBots [9], B&R ACOPOS 6D [10] or Beckhoff XPlanar [11]. These systems then form the core of advanced intralogistics solutions such as SOMIC CORAS [12], shown in Figure 1.2, and Provisur FMS [13], allowing a single system to simultaneously perform transportation, collecting, orienting and grouping tasks which would otherwise require multiple dedicated or complex stations.

The high degree of flexibility of these systems however makes their manual configuration time-consuming and requires considerable human expertise. Especially if the product composition and configuration is adjusted frequently or must adapt dynamically to the current conditions, this manual approach is economically infeasible. For this reason, academia and industrial research have produced many performant, highly automated and distributed controllers capable of autonomously managing large warehouses or industrial complexes without human intervention [7]. In practice however, many factories rely on centralized Programmable Logic Controllers (PLC), adhering to proven technologies and standards [16], such as IEC 61131. PLCs are designed for robustness and reliability, derived from controllers operating on simple logic systems. In this, they are slow to adapt capabilities required for the operation of many state-of-the art algorithms [17]. Additionally, most existing legacy systems do not have the performance required to perform these computationally expensive tasks in real-time, which is required for the reliable operation of the factory.

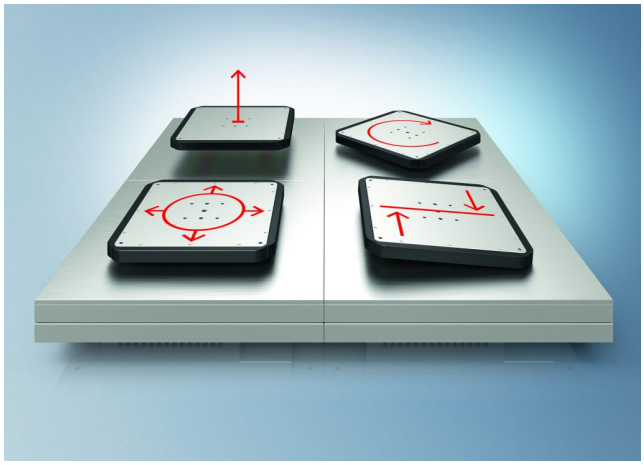


Figure 1.1 Visualization of the six degrees of freedom of Beckhoff XPlanar and similar linear magnetic movers [14].

Figure 1.2 Potential use case of the SOMIC CORAS collecting and grouping system, using mounted containers for transporting products [15].

1.1 State of the Art

The drive away from traditional, centrally controlled factories has led to the advent of Cyber-Physical Production Systems (CPPS) in many industries, defined according to Monostori [18] as autonomous elements of a production system, cooperating with each other in situation dependent ways. In this, they serve as digital representatives of physical production related objects [19], allowing real-time data acquisition and process optimization, such as self-configuration, fault detection and compensation [20] and predictive knowledge-based maintenance [21]. As production systems are typically designed-to-order with accordingly long life times, they must be designed with the ability to continuously evolve and adapt to changing requirements [22]. With software having a significantly shorter product life cycle than hardware, most of these modifications are naturally focussed on the control software. In combination with the increasingly complex requirements for additional control and communication features [23, 24], measures are required to keep verifiability and maintainability at economically feasible levels. While modularization and code reusability using encapsulation and object-oriented approaches are well-established and commonplace in pure software applications, this is far less common in CPPS relying on software developed for IEC 61131-3 compliant PLCs [25]. Despite the drive towards an object-oriented mentality having existed for some time [26], a coordination of the various disciplines involved in the creation of such systems, most of which have developed their own design methodologies, engineering methods and tools, still poses an open challenge [1]. The contribution of this work approaches this by proposing a reusable control software, with all considerations to the physical environment and task encoded in an external, structured and autonomously generated solution, allowing both the initial and re-configuration of the station to be performed with little manual involvement.

Autonomous navigation, and specifically the employment of AGVs in industry, has been a highly relevant topic for decades [27], with a corresponding amount of related works and research. In this, most work relating to controlling agents within confined and well-regulated areas such as factories relies on a, either physical or virtual, guide path approach, designating a graph of navigable paths to be traversed by the actors [7]. This reduction of the problem allows it to be approached using graph-based methods and solvers for scheduling and path finding. The simplest AGV networks, with little computational complexity, typically consist of non-overlapping, single vehicle loops [28] and are limited in their achievable throughput. Alternative approaches rely on a small number of unidirectional paths, with the interaction points placed on dedicated spurs along the path to avoid an interacting vehicle from blocking following vehicles [29]. As spurs require available space adjacent to the paths, such a system is feasible in large factories with

dedicated hallways and areas for machines, but not suited for the presented use case, which operates in a dedicated space with the dimensions of the agents being a significant portion of the overall field size.

Instead of using sparsely connected graphs, most contemporary approaches rely on well connected roadmaps, typically arranged in rectangular grids [30], providing redundancy while allowing agents to follow more direct paths to their destination. Collision-free navigation on these graphs requires solving a Multi-Agent Pathfinding (MAPF) problem, simultaneously generating collision-free trajectories for a number of agents. As the pathfinding problem for a single agent has been well-explored in academia, a number of these approaches have been adapted to apply to multiple agents, such as Cooperative A* (CA*) and its variants Hierarchical Cooperative A* and Windowed Hierarchical Cooperative A* [31]. This approach extends the search space of traditional A* [32] with a time dimension, allowing agents to occupy the same space at different points in time. This allows a serial path finding for multiple agents, each reserving their required space-time cells in a grid, which then acts as an impassable transient obstacle for any following agents. The approach however does not scale well to the number of agents. Specifically, conflicts occurring when multiple movers attempt to navigate the same space at the same time, such as at constrictions with high throughput, must be resolved efficiently for a reliable application of these methods.

A number of algorithms addressing this have been proposed, such as Safe Interval Path Planning [33], which reduces the size of the search space by replacing discrete time steps with contiguous time intervals of collisions and safe periods. An alternative approach is Conflict Based Search [34], which finds the optimal solution of a MAPF problem by wrapping the generated optimal paths of individual agents in a high-level constraint tree. This approach can then efficiently resolve all conflicts by iteratively adding new constraints to the tree until all conflicts are resolved and a valid solution is found. This approach has been further refined, such as in [35], which allows local conflicts between a subset of agents to be resolved more efficiently by grouping them into a meta-agent, separating their search space from the global search space. Furthermore, extensions allowing for continuous time resolution instead of discrete unit time steps have been proposed [36]. These algorithms are however designed to uniquely match a task with a fixed goal to an agent. While this use case is reasonable for most problem statements, it is not optimal for this work, as products of the same type can be considered interchangeable and thus restricting each agent to a single, fixed goal is too strict.

Futhermore, the operation of large fleets of AGVs is typically a lifetime problem, with new jobs coming in during the runtime of the system. For this, an additional layer of scheduling is necessary, assigning available agents to newly arrived jobs. Consequently, new paths must continuously be found and assigned during runtime, avoiding collisions with other agents while minimizing traversal times or keeping deadlines. A number of algorithms expanding on existing MAPF solvers have been proposed to address this, combining the path planning with a dynamic scheduling algorithm. One such approach by Blumenstein et al. [8] is based on the Module Type Package concept and introduces a method for expanding PLC-based modular logistics systems by including flexible transport processes based around AGVs. In this, they establish a novel standardized integration mechanism to minimize the effort necessary to integrate such a system with surrounding stations. A significant difference to the previously discussed algorithms is that they require the system to be capable of choosing between multiple potential destinations. Furthermore, they define transport orders as means of stations to request a product, which can either be initiated as a push from the providing station, or as a pull from the receiving. While such transport orders are a suitable approach in environments where the orders can not be sufficiently predetermined, this results in agents only approaching the order start node after the order is placed. An application of this method to the problem statement discussed here therefore requires careful scheduling of the orders to ensure the efficient flow of empty agents towards their loading points. Their presented interface however is a promising candidate for a standardized integration of the presented controller, as all interactions with surrounding stations remain abstracted throughout this work.

An alternative algorithm for the lifetime control of AGVs has been presented by Ma et al. [37], who expand on the principle of the shared space-time cells used in CA*. In this, free agents pick open tasks to execute

following a fixed order. Open tasks may only be picked if no other agent currently holds a task involving either the start or goal of the open task. As agents remain stationary if they have no task or viable path, such an assignment must be avoided as it has the potential to result in a deadlock of the system. This approach however is only feasible if each start or goal has a relatively small throughput, as it requires tasks to be performed sequentially, only allowing an agent to accept a new task involving the same node after the agent assigned to the previous task has left it. This limitation is acceptable in environments such as warehouses, which typically require the transport of specific products between shelves or to interaction points at the outside of the field, resulting in a large number of potential task start and goal positions, each with a relatively low throughput. The task presented in this work however makes use of only a few such positions, each of which with a significantly higher throughput, making such a limitation infeasible.

1.2 Related Work

The offline solution generation presented in Chapter 3 generates a system of fixed lanes, traversable by the movers and adhering to the validity requirements introduced in Chapter 2. As these requirements concern themselves with both the connectivity of the system and the physical layout of the lanes, this problem can be formulated as the generation of a graph, whose planar layout must fulfil the specified geometric requirements. In this, both the graph topology and placement of the corresponding lanes must be optimized simultaneously. A comparable problem statement is well-known in the area of system design, which deals with with the simultaneous packing of components and routing of required interconnections subject to a number of physics-based considerations. Over the past decades, this has resulted in a number of proposed computational approaches, including rule-based, gradient-optimization and genetic algorithm methods [38].

Lee [39] approaches this as a simultaneous optimization problem, considering geometric, packaging and system functionality properties such as heat flow and pipe lengths. Their work presents a computational environment, allowing the optimization of layouts considering these factors. Due to the non-smooth constraints in their optimization function, they make use of a genetic algorithm optimization, capable of operating without derivatives. Their initial layout however relies on a known feasible arrangement, or alternatively searching the design space in a not further specified way until a feasible layout is found.

Recently, Peddada et al. [40] have proposed a method for efficiently addressing this search. Their presented two-stage design framework consists of separating the optimization problem into the generation of unique topological layouts, which are then optimized using gradient-based approaches formulated from the remaining continuous physical requirements of the system. They define the topological equivalence of two layouts as the ability to transform all contained interconnects between them continuously, without tearing or glueing. Figure 1.3 visualizes this, with the interconnects being represented by the black and red lines. This distinction, following the assumption that gradient-based optimization methods cannot navigate discrete changes in geometric topology, aids significantly in structuring the search space. By first finding a suitable unique topology, a gradient-based search is more likely to quickly converge to a feasible solution as they have demonstrated in [41]. A number of significant differences however exist between the problem statements of their and this work. While a system design has a number of predetermined components and connections, the lane graph initially only dictates the positions of stations, while all other components are generated during runtime to address the validity requirements, namely the splitting and merging of lanes, intersections and corners. Additionally, while their approach considers relatively thin interconnects, the lanes of the solutions developed in this work make up the majority of the overall footprint of the system. Considering that most stations are placed at or near the border of a field, this makes their approach to immediately require the planarity of the generated graph infeasible, resulting in the introduction of intersections to the graph generation of this work. Nevertheless, their overarching concept of separating the topology generation from the layout optimization has proven applicable.

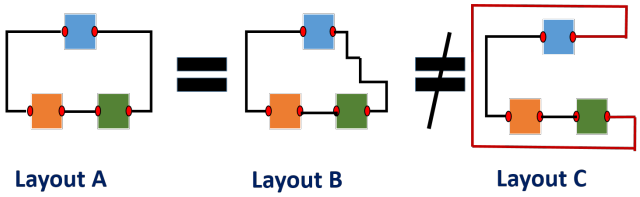


Figure 1.3 Equivalent and distinct geometric topologies of 2D system layouts. The red interconnect of Layout C cannot be continuously transformed into its equivalent in Layouts A or B. Excerpt from [41].

This approach allows the placement optimization, which is discussed in Section 3.3, to make use of a gradient-based algorithm. While optimizations in the context of system design typically deal with objective functions based on multiple physical properties of the components, the minimal approach used in this work only concerns itself with their placement. A similar problem can be found in the visualization of graphs, which has been approached by Eades [42] using a spring-embedder model, placing attracting forces on connected nodes while repelling all others. This method has been improved by Fruchterman and Reingold [43], whose work includes the limitation of the extent of the graph to a frame, removing any translation normal to the boundary of nodes forced against it. Furthermore, they introduce an upper separation limit, outside of which no interaction occurs, improving the performance of the optimization over Eades approach, which computes an interaction force between all nodes, despite the magnitude of the result being negligible for large distances. For this, they propose the grid-square algorithm, reducing the number of combinations which must be tested for their separation, reducing the time complexity of the solver. While in the visualization, the edges of a graph have a negligible thickness, the lanes of the generated solution have a finite width and must be placed in a planar arrangement without any overlaps. Furthermore, as lanes may be placed directly adjacent to each other, the presented approach, discussed in Section 3.3.1, makes use of a hard validity boundary and force cutoff at the border of the footprint. From this follows that the upper limit, while only being a reasonable approximation in the work of Fruchterman and Reingold, is equivalent to comparing all combinations in the presented use case.

1.3 Contribution of this Work

While a number of viable approaches for controlling AGVs in industrial environments exist, the problem discussed in this work has special requirements to the efficiency of the controller, while offering a number of simplifications compared to a more general approach. Namely, these are the small number of interaction points and the variability in the destination of each task, tied to the type of the transported product. Furthermore, the intended deployment on a IEC 61131 compliant PLC must be considered, as well as the need to efficiently reconfigure the system to accommodate the changing requirements of the customer.

This work presents a method capable of offloading most of the computational expense of controlling a fleet of movers operating on a densely populated area into an offline preprocessing step, autonomously generating a solution which is then used to configure a real-time controller without the need for further modifications. Specifically, this method is applied to product distribution tasks found in the intralogistics of packaging systems, which are required to transport products of different types from incoming to outgoing feeds while sorting them according to some given output product type sequence. This is achieved by generating a graph of unidirectional lanes traversable by the movers, making use a separation of unique topological layouts and an iterative force gradient based optimization. Furthermore, the concept of item sequences is introduced as a method to efficiently ensure a valid arrival order of movers at the dropoff points, without requiring the controller to otherwise consider the positions or actions of other movers when making a routing decision.

The remainder of this thesis comprises five further chapters. Chapter 2 formalizes the problem statement and underlying model, before introducing the components of the presented solution process and discussing a number of design strategies to be followed. Afterwards, Chapter 3 introduces the offline preprocessing pipeline, which is executed prior to the deployment of the system. The corresponding real-time controller is presented in Chapter 4, in addition to a simulation environment which can be used to verify the reliable operation of the system in software. Finally, Chapter 5 discusses the performance of the system on a number of exemplary problem statements, while Chapter 6 concludes this work by evaluating the fitness of the presented methods to solve the stated problem and by elaborating on potential future developments.

2 Problem Statement and Solution Design

This chapter outlines the purpose and requirements of the system to be designed. These requirements take current industrial applications and practices, the capabilities of available hardware, and expected future use cases into account. Afterwards, a set of models representing the system are defined, on which the components of the presented solution are based and justified. Finally, a set of design strategies is formulated to aid in creating an algorithm producing efficient and reliable solutions.

The system is intended to be used as part of a larger assembly, with the purpose of rearranging products while transporting them from a preceding to a following subassembly. For this, products of different types are inserted into the system at one or more input feeds and must be distributed to one or more output feeds. Products of the same type are interchangeable, individual products do not have an explicit destination. If a feed provides or receives products of different types, it must abide to a given type sequence as defined in Section 2.1. This task is to be accomplished using a planar motor system, consisting of a field and a number of movers. The field is formed by a flat area of tiled electromagnetic stators, while each mover is an individually controllable platform on which a bracket specific to the transported products is mounted. Interactions with the feeds occur at stations, designated areas of the field in which products are loaded or unloaded between feed and mover. This is accomplished through a combination of external hardware and movements performed by the mover, which must be synchronized between all involved subassembly controllers. Figure 2.1 visualizes an example of such a system.

A number of additional requirements may be formulated for a specific application to ensure reliable and deadlock-free operation or a minimum sustained throughput. The system may need to provide sufficient redundancy to accommodate for a variance in timing of products on incoming feeds. Similarly, feeds may only have a limited buffer size, resulting in a maximum time for which a station is able to delay operation before another mover must be available. One significant advantage of such a system over traditional solutions is the ability to configure product flow in software, without the need to modify any hardware of the machine. To take advantage of this however, the system must be able to efficiently compute solutions for various configurations while working with the initially laid out hardware. Specifically, this includes the shape and size of the field, the positions of stations, and the number and capacity of available movers.

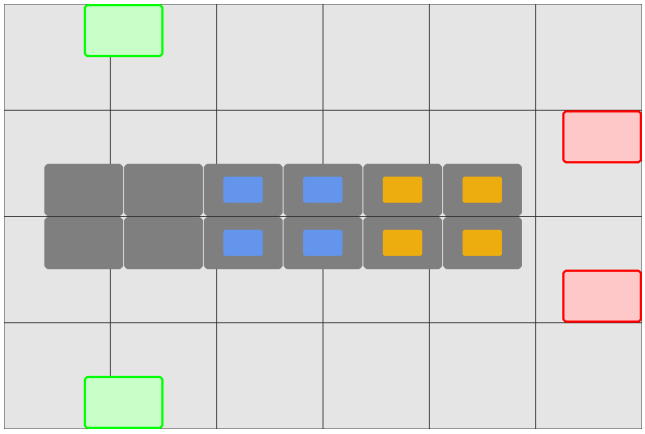


Figure 2.1 Example of a six by four tile environment with four stations and twelve movers.
The footprint of provider stations is highlighted green, that of receiver stations red. Movers carrying an item are marked with colored rectangles, each color representing an item type.

2.1 Model Definition

This section defines the models used to formalize the outlined system specifications and requirements. Specifically, the representation of elements on the field, the dynamics of the movers and a generalization of the station interactions and transported products are elaborated. All specifications relating to hardware capabilities used for calculations throughout this work are based around the manufacturer specifications of the Planar Motor Series 3 [44], B&R ACOPOS 6D [10] and Beckhoff XPlanar [45].

The field and all mover actions are represented in a two-dimensional coordinate system. While planar motor systems allow for some three-dimensional movement, they are generally not sufficient for movers to pass over each other without colliding and are therefore omitted here. While the field may take any shape and dimensions, a typical application consists of tiled stators forming an obstruction-free field. As discussed in Section 3.1.3, the offline preprocessing algorithm requires a convex field with the stations placed at the field boundaries. Due to the tiled nature of the field, it is therefore assumed to be rectangular.

Movers are modeled as axis-aligned rectangles, with all movers assumed to be of equal size and capabilities. Their dynamics are modeled using a constant acceleration and symmetric deceleration, which is defined as $a_{\max} = 20\,\mathrm{m\,s^{-2}}$. Their maximum achievable velocity is defined as a constant $v_{\max} = 2\,\mathrm{m\,s^{-1}}$. With manufacturers claiming a $< 5\,\mu\mathrm{m}$ repeatability, movers can be controlled precisely enough to allow any required separation to be neglected throughout the discussions of this work. Movers navigate the field following lanes, chains of fixed waypoints connected by straight sections. This closely represents the waypoint operation of the planar motor system controller interface, which is further elaborated in Section 4.1. While movers are generally able to perform continuous trajectories such as arcs, the presented model exclusively makes use of linear point-to-point movement. This consequently requires movers to come to a complete stop when changing directions, but simplifies control and allows for densely placed lanes on the field, as corners are not cut. The presented algorithm generates a solution graph of nodes representing coordinates on the field, connected by unidirectional edges the movers can traverse.

Each feed entering or exiting the system is represented by an interaction station. They serve as input and output interfaces to adjacent systems and are modeled as areas with a fixed position and rectangular footprint. The footprint of the station represents the space used by the mover during interaction and must be free of other movers during this time. During the offline preprocessing and simulation, the interaction is abstracted. A mover interacts with a station by moving to the center point of the footprint and remaining stationary for an interaction duration τ_I, before being returned to the controller. Depending on the requirements, τ_I is a fixed value or distributed randomly within some bounds. Lanes connecting to the station must pass through an interface, a fixed waypoint adjacent to the station footprint. Interfaces serve as waiting and staging areas, optimizing throughput as discussed in Section 2.3. For the same reasons, interfaces serve exclusively as unidirectional sinks or sources of the station. Footprints of stations cannot overlap, as stations operate independently of each other and serve one mover at a time. Additionally, interfaces cannot be shared between stations. While the station positions may be optimized when initially laying out the system, any modifications to the hardware during operation are not possible. For this reason, the positions are specified in the problem statement and are not optimized by the presented algorithm.

The products transported by the system are abstracted as items. Items are indivisible and each mover carries one item at a time. Furthermore, all items of the same type are interchangeable. Item types represent different mover loading states, allowing movers carrying no products to be regarded as carrying an "Empty" type item while other item types may represent different product types, amounts, or combinations of both. Following this model, stations modify the item type of the interacting mover. Each station requires and provides at least one item type. If the station is part of a multi-stage operation such as partial loading, unloading or manipulation, each stage can be represented by a different item type. This work is limited to provider and receiver type stations with fixed, cyclic sequences. Sequences are finite tuples of item types T which must be followed in the given order. If the end of the tuple is reached, the sequence continues from the beginning of the tuple. Providers sink empty movers and source movers following the sequence, while receivers must be supplied movers following the sequence and source empty movers.

2.2 Solution Structure

This section defines the environment, network, graph and item sequence structures used throughout the presented algorithm. Initially, the problem statement is formalized as an environment. A network representation is then used during offline preprocessing discussed in Chapter 3, containing all generated information. The preprocessing outputs a reduced solution graph, which only retains the information required by the controller presented in Chapter 4. Finally, item sequences improving the efficiency of the controller are introduced.

2.2.1 Environment

The environment encodes the problem statement and serves as input to the presented offline preprocessing algorithm. It contains all data necessary to generate a viable solution. This includes the size and shape of the field, typically in the form of a stator tile arrangement and information about any obstacles. Additionally, the size and number of the movers is given. The algorithm can be permitted to reduce the number of movers if doing so improves the solution performance. Otherwise, the exact number of mover must fit on the field at all times. All interaction stations are specified, stating their position, footprint size and flow specifications. In this work, flow specifications consist of the type of the station, either serving as provider or receiver, a tuple of item types representing the cyclic station item sequence, and an expected flow rate for each item type. Additional information such as the interaction duration and shortest and longest time between interactions can be included for use by the controller.

In order to provide a solution to an environment, there must exist sufficient connections from providers to receivers and vice-versa to balance the expected flow for each item and station. A network providing these lanes is then defined as well-balanced. As discussed in Section 2.2.4, this is a necessary, but not sufficient criterion for a reliable operation.

2.2.2 Networks

Networks represent the lane structure on the field. They consist of a number of primitives, serving as waypoints, connected by arcs. Chains of arcs are grouped into paths and sections, which are used to track additional information over larger parts of the field. Networks contain all information generated in the offline preprocessing pipeline, with each step modifying the structure or primitives of the solution network. The goal of the offline preprocessing is to generate a valid solution network, for which it must fulfil the environment flow specifications while adhering to a number of design requirements detailed below. Figure 2.2 shows an example of a valid network, which will serve as a visualization throughout this chapter. For this network, the flow specifications can be fulfilled by providing a path from either provider to both receivers, while only requiring a return connection to a single provider from each receiver.

Primitives

Primitives are unique parametric building blocks of the network, representing two-dimensional rigid elements. Their footprint is rectangular and aligned with the coordinate system axes, typically outlining the mover size by the chosen collision tolerance. This approach allows for a simple overlapping detection between primitives to guarantee sufficient clearance between movers at all times. Consequently, a network is only valid if no two primitives overlap. During optimization, primitives are moved on the field to achieve an improved solution. Some primitives, such as stations or interfaces, are fixed in place.

There are multiple types of primitives, each with a set number of incoming and outgoing arcs that must be fulfilled for the network to be valid:

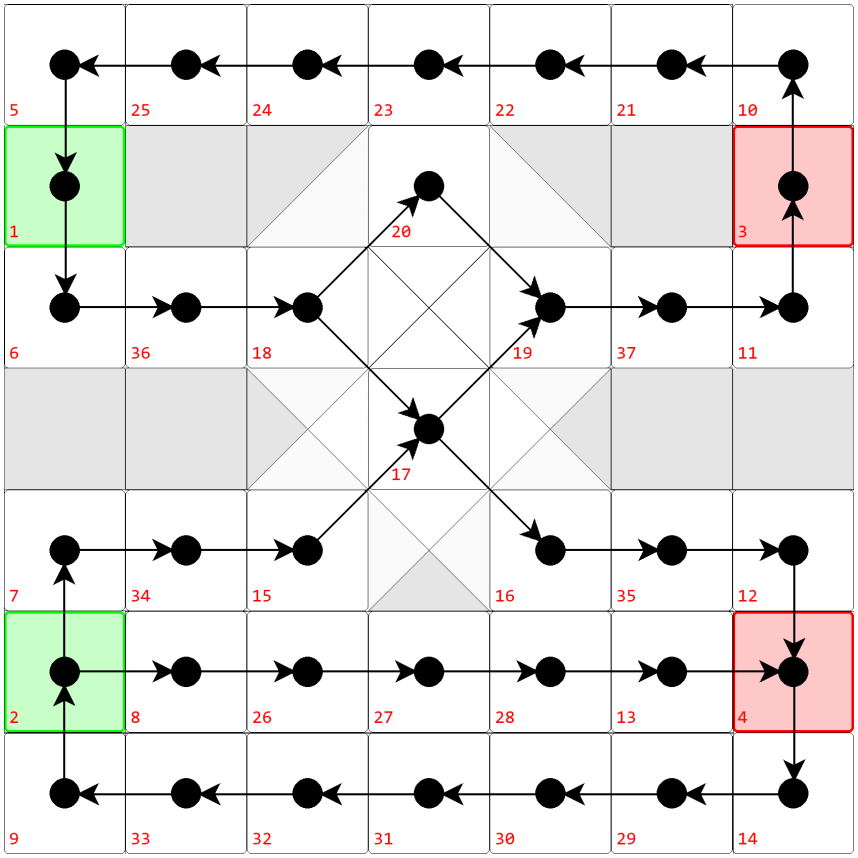


Figure 2.2 Example of a valid network. The contained primitives are numbered for better reference, such as the providers #1 and #2 and the receivers #3 and #4.

- **Stations** represent the interaction stations defined in Section 2.1. As their position is given in the environment, they are fixed on the field. The primitives #1 and #2 in Figure 2.2 are providers, #3 and #4 receivers. They require at least one incoming and one outgoing arc to interface primitives while, as discussed in Section 2.4, a third connection is possible in some circumstances.
- **Interfaces** are fixed adjacent to their station. They have one incoming and one outgoing arc, connecting to their station and one other primitive. #5 through #14 are interfaces.
- **Corners** join two straight lanes at an angle, connecting one incoming and one outgoing arc. #15, #16 and #20 are corners. They are similar to interfaces, but are subject to optimization during the offline preprocessing, where they may be moved, added or removed.
- **Splits** represent a splitting lane, connecting one incoming and two outgoing arcs as for #18. They are introduced during the network initialization if a primitive has too many outgoing arcs.
- **Merges** represent two merging lanes, connecting two incoming and one outgoing arc as for #19. Like splits, they are introduced if a primitive has too many incoming arcs.
- **Crosses** are the intersection point of two lanes, each consisting of one incoming and one outgoing arc. Movers cannot change lanes while passing the primitive, they must travel along paired arcs. #17 is a cross, intersecting the lanes #15 $\rightarrow$ #19 and #18 $\rightarrow$ #16.
- **Waits** are only employed in the post-processing step of the pipeline, splitting long arcs to serve as valid, but optional, waiting points for movers. The incoming and outgoing arc resulting from the split are always parallel. All primitives #21 and above are waits.

Arcs

Arcs represent a straight-line movement between a tail and head primitive. Their footprint spans the area covered by the mover during traversal, including the footprints of both tail and head primitives. This results in a four or six-sided convex polygon, dependent on the relative position of the primitives. In a valid network, arcs cannot overlap primitives that are not their tail or head. While this condition is sufficient in practice as discussed in Section 3.3.1, a network is only valid if additionally no two arcs that do not share a tail or head primitive overlap. This implies that the two-dimensional representation of the network cannot contain crossing arcs, instead requiring some form of intersection. The arrangement in Figure 2.3 shows a typical example where this additional condition is required, as neither primitive is in direct violation but the network is visibly overlapping. This can occur during iterative placement optimization, if primitive #2 passes through arc #4 $\rightarrow$ #5 without being caught by the collision detection due to the discretization error.

The requirement of non-overlapping primitives implies a minimal length of each arc, coinciding with the index length l elaborated in Section 2.3.1. Requiring a longer minimal length, typically an integer multiple of l, can be used to guarantee additional waiting points along the length of the arc. This may be beneficial to the robustness of the solution, as additional capacity for movers along an arc can prevent following stations from being undersupplied or preceding elements from being blocked if the mover flow rate fluctuates too much. On the contrary, permitting two primitives connected by an arc to overlap may aid in finding a valid or improved solution in a tight environment. An example for this is shown in Figure 2.4, in which alternative networks offering the same connectivity, such as a cycle, perform worse for some environments. In this case however, additional care must be placed on configuring awaitable positions in the solution graph, as not all primitives can be populated by a mover at the same time without colliding.

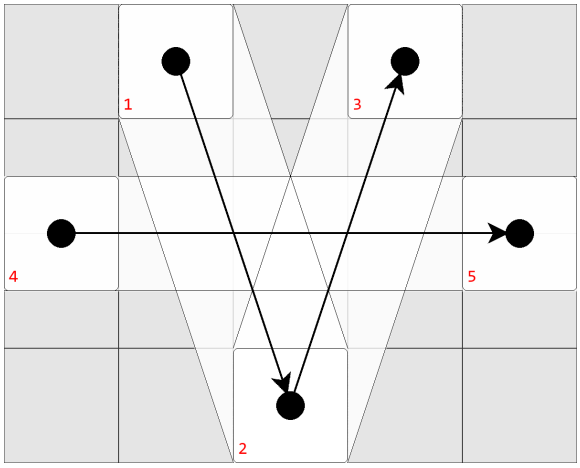


Figure 2.3 Example of an invalid arc arrangement despite no primitives being in direct violation, as arcs that do not share a tail or head primitive overlap.

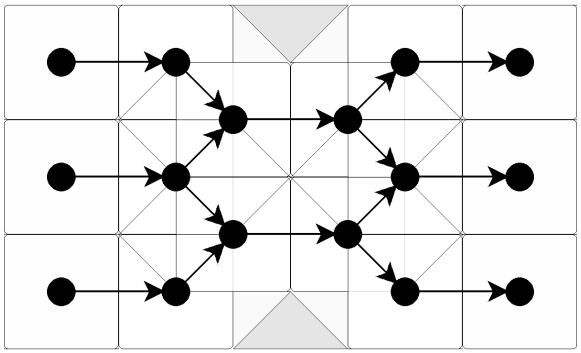


Figure 2.4 Example of a network benefiting from short arcs. Most alternative solutions for the presented connectivity and given space are less efficient.

Paths

Paths represent a continuous chain of arcs, connecting two station primitives. For a path P, the path start primitive $\mathrm{Start}(P)$ is the tail of the first arc of the contained chain, while the path goal primitive $\mathrm{Goal}(P)$ is the head of the final arc. Paths are initially derived from the environment flow specifications as described in Section 3.1. There, one path per direction is created for each pair of stations if some flow between the stations is expected. For a path P, the set of managed item types $I_M(P)$ then contains all item types which have an expected flow from the start to the goal station. While the contained arcs may be modified during preprocessing, any item flow between two stations will always follow the connecting path. As all arcs are contained in one or more paths, this allows for a simple computation of the expected flow for each item type along an arc.

Sections

Sections represent a continuous chain of arcs, connecting a start and goal primitive. For a section A, $\mathrm{Start}(A)$ and $\mathrm{Goal}(A)$ are defined as for paths. All and only primitives of type station, merge and split are section starts and goals. As shown in Figure 2.6, this results in a primitive-section graph reduced from the primitive-arc network. Crosses may be contained in two sections, one along each pair of arcs, while interfaces, corners and waits are contained in a single section. From this follows that the order of movers within a section is fixed, they exit a section in the same order they entered. This property is used in the controller in combination with item sequences to ensure the adherence to station item sequences. For two sections A and B

$$A \text{ precedes } B \iff B \text{ follows } A \iff \mathrm{Goal}(A) = \mathrm{Start}(B)$$

Each arc is contained in exactly one section. From this follows that a path consists of one or more chained sections, while each section is part of one or more paths. A path also contains all arcs of a contained section. For a section A, the set of managed item types

$$I_M(A) = \bigcup I_M(P) \; \forall_P : A \in P$$

is defined as the union of all sets of managed item types of the paths containing the section.

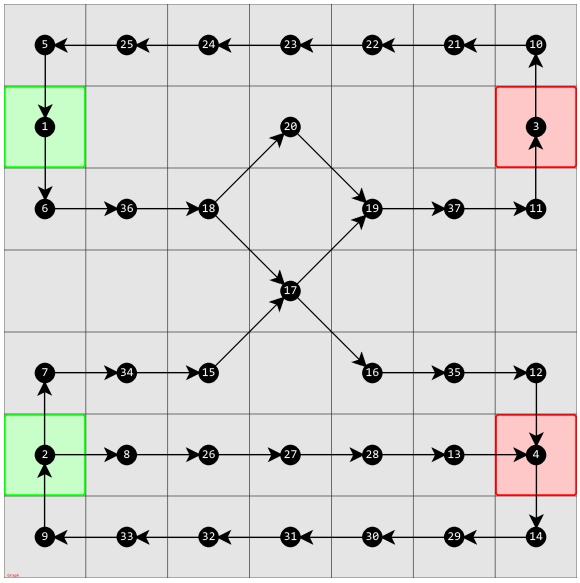


Figure 2.5 Reduced solution graph of the network in Figure 2.2. All footprints have been omitted, as they are not required by the controller.

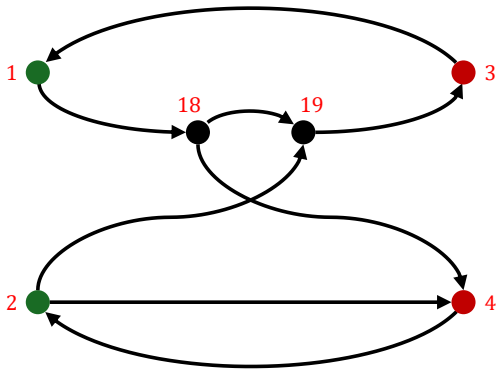


Figure 2.6 Primitive-section graph of the network. Note that the cross #17 is not a section start or goal and therefore not a node in this graph.

2.2.3 Graphs

Graphs are static, reduced versions of networks, containing only data necessary for controller operation as shown in Figure 2.5. A solution graph is generated from the final network at the end of the offline preprocessing pipeline, which is then used in the live environment. Graphs consist of nodes connected by edges.

Nodes are generated from primitives. They do not have a footprint, but represent a single position on the field to be used as a waypoint for mover navigation. Instead of the primitive type information, nodes have awaitable and skippable properties to be used by the controller. A skippable node may be passed by a mover without decelerating. This is only possible if the incoming and outgoing edge are parallel, nodes connecting edges at an angle cannot be skipped. An awaitable node allows for a mover to come to a standstill and not immediately leave the node. Some critical nodes, such as intersections, may not allow this to prevent blocking movers on other paths.

Edges are simplified representations of arcs. They do not have a footprint, retaining only information about their tail and head nodes, which match the tail and head primitives of the arc. Paths and sections are generated from their network equivalents.

2.2.4 Item Sequences

In the physical system, the duration of any operation is subject to some variation. While stations expect a given operation rhythm, it is possible that incoming products are delayed or an arrival time cannot be precisely predicted at all. This results in uncertain interaction durations, which in turn prevents the use of a preplanned, fixed operation schedule and required a dynamic planning of the mover trajectories. The mentioned uncertainties, in combination with the high traffic caused by the densely populated field, make it prohibitively expensive to exactly determine the traversal duration and arrival order of movers ahead of time. This however is required to route movers efficiently while ensuring adherence to the station item sequences, for which movers must follow an order determined by their carried items. Movers arriving at a stations faster than movers scheduled earlier in the sequence must therefore wait for their slot without blocking the approach paths of other movers.

In a typical solution graph, most sections are shared by multiple paths to ensure the required connectivity in the limited available space. To minimize the time for which the controller must plan ahead, it is therefore sensible to delay committing to a destination station and consequently to a set path as long as possible for any one mover. Specifically, decisions limiting the possible destinations or fixing the order of arrival always occur when paths merge or diverge, which is only the case at section transitions.

Using this approach requires a way to efficiently determine whether any routing decision results in an invalid system state, a set of mover positions and items in which all possible further decisions lead to an eventual violation of the station sequences. For this purpose, the controller presented in Chapter 4 makes use of item sequences, which efficiently track the set of item types allowed to enter a section at the current state of the overall system. The sequences of any section can be generated from the network structure and the sequences of all following sections. This allows the generation, presented in Algorithm 1 to work iteratively, inheriting sequences from already completed following sections. The algorithm is initialized by assigning sequences to all sections ending in a station, based on the station sequence given in the environment description. In this, the station acts as a zero-length section containing only the provided station sequence. A controller using these sequences can then indefinitely fulfil the required station sequences assuming a well-balanced overall network.

Each item sequence is assigned to one or more sections, while each section contains one or more item sequences. For each sequence s, the set of of currently allowed item types $I_P(s)$ is defined. The set contains the item types that may enter the assigned sections next without leading to an invalid system

state and is therefore dependent on the current state of the system. Additionally, the set of managed item types $I_M(s)$ is defined for the section such that $I_P(s) \subseteq I_M(s)$ for any system state.

If $S_A = \{s_1, \ldots, s_n\}$ is the set of sequences assigned to a section A, then the set of managed item types for the section is

$$I_M(A) = \bigcup_{i=1}^{n} I_M(s_i)$$

Additionally, the set of currently allowed item types for the section is

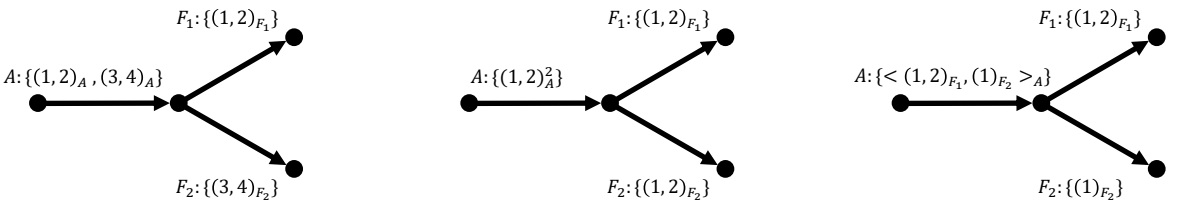
$$I_P(A) = \bigcup_{i=1}^{n} I_P(s_i)$$

A mover M carrying an item of type I may only enter A at $\mathrm{Start}(A)$ if $I \in I_P(A)$. When M is scheduled by the controller to enter A, M must commit to one allowed sequence $s_c \in \{\, s \in S_A \mid I \in I_P(s) \,\}$. Upon commitment, s_c is then advanced to account for the new state of the system, potentially altering $I_P(s_c)$. This scheduling fixes the order in which movers can traverse the section, M must wait to enter A until all movers scheduled earlier have entered A.

For a section start primitive G, the set of outgoing sequences S_{out} contains all sequences assigned to the sections outgoing from G. If S_{out} can be separated into multiple disjunct subsets $S_{\mathrm{out},1}, \ldots, S_{\mathrm{out},n}$ while not sharing any managed item types

$$I_M(s) \cap I_M(t) = \emptyset \; \forall s \in S_{\mathrm{out},i}, \forall t \in S_{\mathrm{out},j} \; \forall i,j \in \{1, \ldots, n\}, i \neq j \tag{2.1}$$

each subset of S_{out} can be inherited separately to the incoming sections of G as shown in Figure 2.7a. The order of movers between subsets is irrelevant for fulfilling the individual sequences.



(a) Independent sequences **(b)** Same sequences **(c)** Shared sequences

Figure 2.7 Examples of different cases of sequence inheritance at split primitives.

Single Sequence

Single sequences contain a single section and represent a single item type. A sequence s containing the item I consequently has

$$I_P(s) = \{I\} = \boldsymbol{const.}$$

Single sequences are used as target of an inheritance whenever all S_{out} are single sequences or the contained section only carries one inherited item type. Multiple single sequences of the same item type contained in the same section are redundant, additional ones can be omitted. s is then notated as $(I)_A$ if s is assigned to the section A.

Multi Sequence

Multi sequences contain a single section and represent a static, cyclic item sequence. For a multi sequence s representing the tuple of item types $T = (I_1, \ldots, I_n)$

$$I_P(s) = \{I_k\} \quad \text{for} \quad 1 \leq k \leq n$$

Initially $k = 1$, increasing by 1 with each sequence commit until wrapping around when reaching n. s is then notated as $(I_1, \ldots, I_n)_A$ if s is assigned to the section A. If multiple independent sequences assigned to the same section are equal, they are interchangeable as shown in Figure 2.7b. The sections are then notated as $(I_1, \ldots, I_n)_A^n$ for n independent sequences.

Split Sequence

Split sequences contain a single section A and have either a split primitive as goal or precede another split sequence. Split sequences are required when a section nontrivially splits up into multiple sections. A split is trivial if all sequences in S_{out} are of type single or of type multi and represent the same item sequence or S_{out} only contains a single sequence. Sequences in trivial splits can be inherited by a set of sequences of the same type, whereas naively inheriting a non-trivial set results in an unsafe controller as demonstrated in Figure 2.8.

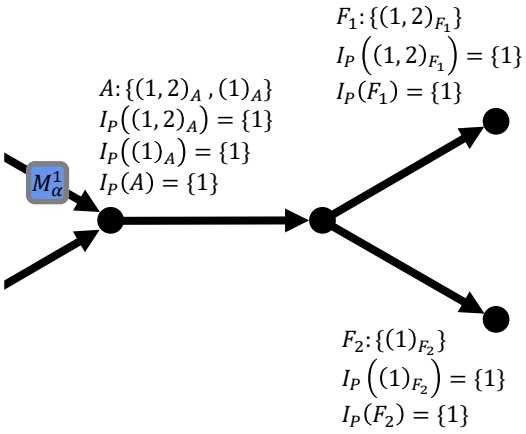
The presented control sequence could be avoided by committing a mover not only to A, but F_1 simultaneously. This however violates the principle of delaying any decisions as long as possible, decreasing the overall efficiency of the system. The presented controller makes use of an alternative, token-based approach. Whenever a mover carrying an item of type I is committed to the split sequence, a token T_I is created. When the mover commits to a following sequence in S_{out}, T_I is removed again. A mover is only allowed on the section if the number of tokens of the item type in existence is smaller than the number of items of that type accepted by the sequences in S_{out} in the current system state. Then, the set of currently allowed item types for a split sequence s is defined as

$$I_P(s) = \{\, I \in I_M(s) \mid \#T_I(s) < T_{I,\max}(s) \,\}$$

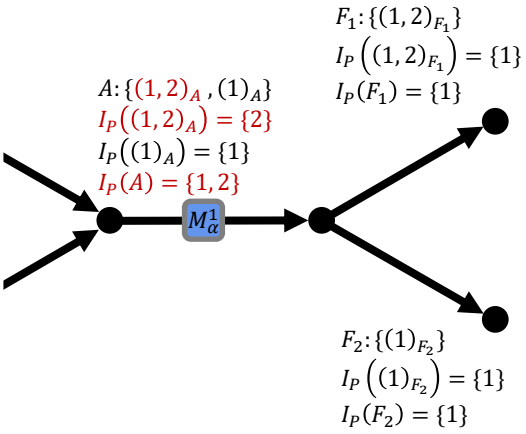
using

$$T_{I,\max}(s) = \sum_{t \in S_{\text{out}}} T_{I,\max}(t)$$

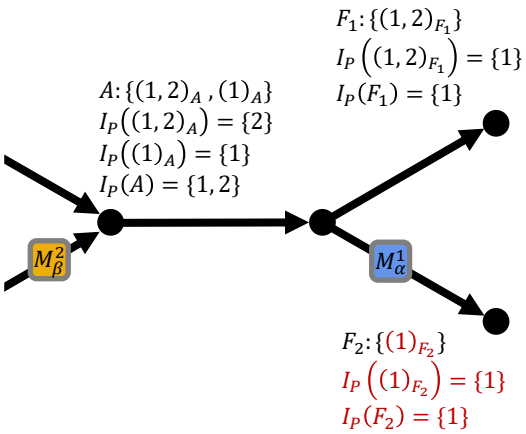
$T_{I,\max}$ depends on the sequence type. For single sequences $T_{I,\max}$ is ∞ if $I_P(t) = \{I\}$, otherwise 0. For multi sequences, $T_{I,\max}$ is the number of elements equal to I in the contained sequence starting from the current index. Merge sequences inherit $T_{I,\max}$ from their original sequence type. s is then notated as $< s_1, \ldots, s_n >_A$ for $s_i \in S_{\text{out}}$ being the following sequences. This approach requires that split sequences are the only sequences able to commit to their inheritors, which must be ensured during inheritance.



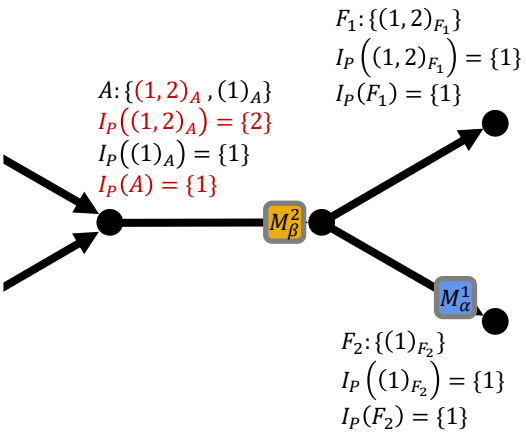
(a) Initially, $I_P(A) = \{1\}$.



(b) This allows a mover M_α^1 carrying an item of type 1 to enter A. The controller commits the mover to the sequence $(1,2)_A$, resulting in a change of $I_P((1,2)_A) \leftarrow \{2\}$.



(c) If M_α^1 however, upon reaching the split is selected to travel along F_2, a commit to $(1)_{F_2}$ would result in a discrepancy.



(d) While a second mover M_β^2 carrying an item of type 2 could then enter A as $I_P(A) = \{1,2\}$, neither $I_P(F_1) = \{1\}$ nor $I_P(F_2) = \{1\}$ would allow M_β^2 to continue, resulting in a blocked path and deadlock of the system.

Figure 2.8 Scenario in which a naively inherited split $S_A = \{(1,2)_A, (1)_A\}$ results in a deadlock of the system. This can be avoided by assigning the split sequence $S_A =< (1,2)_{F_1}, (1)_A >_{F_2}$.

Merge Sequence

If a primitive has multiple incoming sections, all sections need to be considered simultaneously, as a control sequence resulting in an unsolvable state could otherwise occur as demonstrated in Figure 2.10. Each item type passing through the merge is either exclusively provided one by one, or shared between multiple incoming sections. For a set of incoming sections $P = \{P_1, \dots, P_n\}$, the set of shared items is defined as

$$I_{\text{share}} = \bigcup I_M(P_i) \cap I_M(P_j) \; \forall_{i,j} \in \{1, \dots, n\}, i \neq j$$

For each pair of sequence $s \in S_{\text{out}}$ and section $P_k \in P$, the inherited items are

$$I_{\text{inherit}} = I_M(s) \cap I_M(P_k)$$

Then, s is exclusive to P_k with regards to I_{inherit} if and only if

$$I_{\text{inherit}} \cap I_{\text{share}} = \emptyset$$

allowing P_k to inherit s without considering other incoming sections. If I_{share} only consists of one item type a single sequence is used, otherwise s is inherited using a sequence of the same type. If S_{out} only contains a single sequence with nonempty I_{inherit}, it is implicitly exclusive. If non-exclusive sections exist, s is converted to a merge sequence and inherited by the remaining sections.

A merge sequence is spanned over multiple sections forming a tree structure of merge primitives. As shown in Figure 2.9, this results in the sequence having multiple entry points. The exit point is the goal of the root section, outgoing from the last merge primitive. If a sequence is shared between multiple incoming sections, it is converted to a merge sequence if it is not already an inherited merge sequence.

Sequences converted to merges retain their underlying structure, but apply the rules of allowance to all entry points. When a mover is committed to the sequence at an entry point, all edges connecting the entry with the exit point are reserved for this mover using first-in-first-out (FIFO) queues. A mover can only traverse an edge if it is at the head of the queue, popping the reservation while passing. While all primitives contained in the merge sequence are entry points, movers already on a section contained in the sequence may only commit to it if this does not violate the queue. The item sequence is immediately modified as usual on commitment, allowing all movers at each entry point to attempt to match the new set of allowed items and plan a traversal. s is then notated as $(I_1, \dots, I_n)^*_A$.

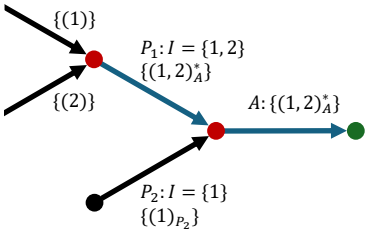
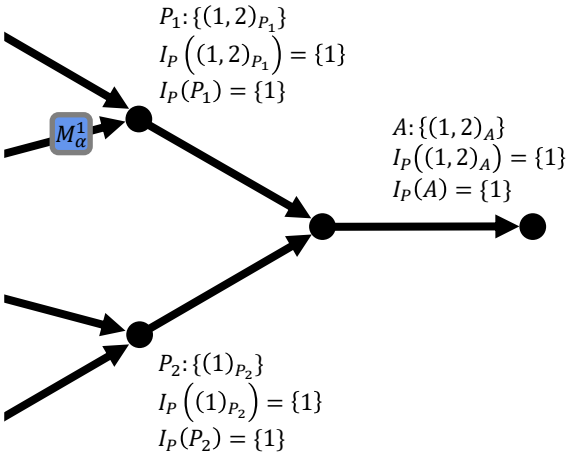
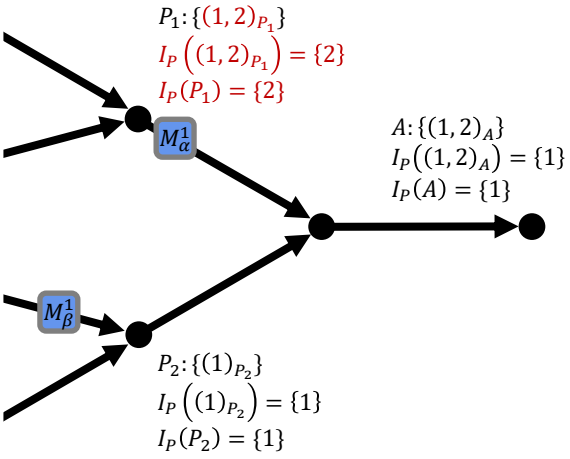


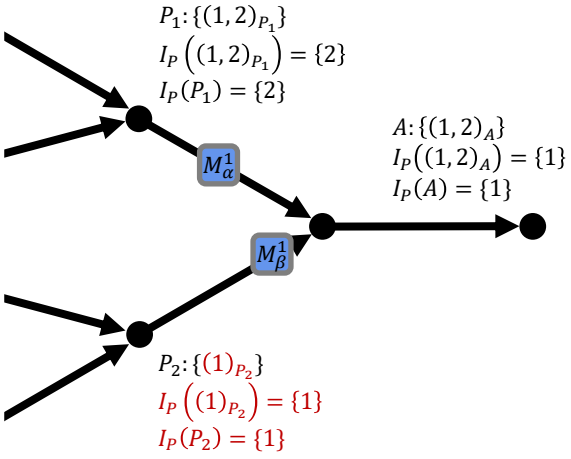
Figure 2.9 Example of a merge sequence, highlighted blue. The exit point is green, the entry points are red.



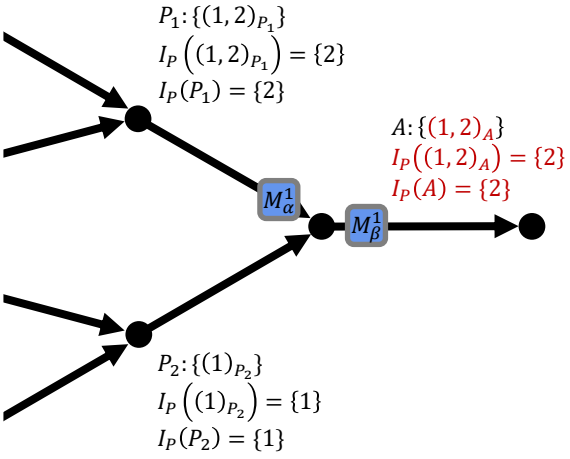
(a) Initially, $I_P(P_1) = \{1\}$.



(b) This allows a mover M_α^1 carrying an item of type 1 to enter P_1. This updates $(1,2)_{P_1}$, resulting in $I_P(P_1) = \{2\}$.



(c) If a second mover M_β^1, also carrying an item of type 1, nears the merge primitive using P_2, it can contest M_α^1 if it reaches the merge earlier and is scheduled to continue first.



(d) Then, M_β^1 is allowed to enter A, as $(1,2)_A$ has not yet been reserved by M_α^1 and $I_P(A) = \{1\}$. This updates $(1,2)_A$ for $I_P(A) \leftarrow \{2\}$, blocking M_α^1 and deadlocking the system.

Figure 2.10 Scenario in which naively inherited merges $S_{P_1} = \{(1,2)_{P_1}\}$ and $S_{P_2} = \{(1)_{P_2}\}$ result in a deadlock of the system. This can be avoided by assigning the merge sequence $S_{P_1} = (1,2)_A^*$ to P_1.

Cycle Sequence

If a network contains a cycle of sections without a station, the iterative inheritance algorithm will eventually reach a deadlock, in which no open section has all following sections populated. In this case, a cycle detection algorithm such as a depth first search is used to determine the set of sections with cyclic dependencies. This set can then be populated by a single cycle sequence. If multiple cycles exist and overlap, generating a sequence for one of the cycles breaks all adjacent cycles, creating at least one open section in which all following sections populated. This section can then be used as a seed to iteratively populate the other sections as usual.

The cycle sequence s combines the token-based approach of the split sequence with the multiple entry points of the merge sequence. A single cycle sequence spans all sections of the cycle, with all primitives with an outgoing section not in the set forming the exit points. The set of following sequences S_{out} then consists of the union of all following sequences of all exit points, while the set of currently allowed item types $I_P(s)$ is calculated as defined above for split sequences. The rules of mover commitment are applied to all entry points simultaneously. Due to the cyclic nature, there exist multiple paths from each entry to each exit point, which makes the mover routing non-trivial. The controller presented in this work prioritizes possibly looping the cycle over waiting at potential exit points to prevent deadlocks. s is then notated as $< s_1, \ldots, s_n >^*_A$ for $s_i \in S_{\text{out}}$ being the following sequences and A a sequence or identifier of the cycle.

Algorithm 1 Iterative item sequence generation and assignment

Input: Sections $N = \{A_1, \ldots, A_n\}$ of the network
Section goals $P = \{P_1, \ldots, P_m\}$ of the network
Station sequence tuples $T_G = (I_1, \ldots, I_{n_G})$ for $G \in P : G$ is STATION
Output: Section-sequence assignments $S_{A_k} = \{s_1, \ldots, s_{n_k}\}$ for $A_k \in N$

Open section goals $P_{\text{Open}} \leftarrow \{\, P_k \in P \mid P_k$ **is** STATION $\,\}$
Complete section goals $P_{\text{Complete}} \leftarrow \emptyset$
while $P \neq P_{\text{Complete}}$ **do**
  if P_{Open} **is not** empty **then**
    INHERITCYCLE()
  else
    $G \overset{\text{pop}}{\leftarrow} P_{\text{Open}}$
    if G **is** STATION **then**
      INHERITSTATION(G)
    else if G has two outgoing sections **then**
      INHERITSPLIT(G)
    else
      INHERITMERGE(G)
    end if
    $P_{\text{Complete}} \overset{\text{put}}{\leftarrow} G$
    $P_{\text{Open}} \overset{\text{put}}{\leftarrow} \{\, \text{Start}(A_k) \mid A_k \in N,\ S_{A_k}$ is populated $\,\}$
  end if
end while

function INHERITCYCLE
  Cyclic primitives and sections $P_{\text{Cycle}}, N_{\text{Cycle}}$ using e.g. depth first search
  Outgoing sequences $S_{\text{out}} \leftarrow \bigcup S_B \; \forall B \in N \setminus N_{\text{Cycle}} : \text{Start}(B) \in P_{\text{Cycle}}$
  All $A \in N_{\text{Cycle}} \overset{\text{put}}{\leftarrow}$ new CYCLE $< S_{\text{out}} >^*_{\text{Cycle}}$ ▷ Note that all sections inherit the same sequence.
  $P_{\text{Complete}} \overset{\text{put}}{\leftarrow} P_{\text{Cycle}}$
  $P_{\text{Open}} \overset{\text{put}}{\leftarrow} \{\, \text{Start}(A_k) \mid A_k \in N,\ S_{A_k}$ is populated $\,\}$
end function

function INHERITSTATION(Station G)
  if G has one incoming section **then**
    $A_{\text{in}} \leftarrow$ incoming section: $\text{Goal}(A_{\text{in}}) = G$
  else
    $G^* \leftarrow$ new MERGE at G, representing the implied merge
    $A_{\text{in}} \leftarrow$ new section $G^* \rightarrow G$ with zero length
    $A^*_{\text{in}} \leftarrow$ incoming sections: $\{\, A \in N \mid \text{Goal}(A) = G \,\}$
    All $A \in A^*_{\text{in}}$ are updated to $\text{Goal}(A) \leftarrow G^*$
    $P_{\text{Open}} \overset{\text{put}}{\leftarrow} G^*$
  end if
  if T_G has one unique item type **then**
    $S_{A_{\text{in}}} \overset{\text{put}}{\leftarrow}$ new SINGLE $(T_G)_A$
  else
    $S_{A_{\text{in}}} \overset{\text{put}}{\leftarrow}$ new MULTI $(T_G)_A$
  end if
end function

function INHERITSPLIT(Split G)
  Incoming section $A_{\text{in}} \leftarrow \text{Goal}(A_{\text{in}}) = G$
  Outgoing sections $N_{\text{out}} \leftarrow \{ B \in N \mid \text{Start}(B) = G \}$
  Outgoing sequences $S_{\text{out}} \leftarrow \bigcup S_B \; \forall_B \in N_{\text{out}}$
  for $S_{\text{out},i} \subseteq S_{\text{out}}$ **do** ▷ Independent subsets using Equation 2.1.
    if only one sequence $s_j \in S_{\text{out},i}$ **then**
      if s_j **is** MERGE **then**
        $S_{A_{\text{in}}} \overset{\text{put}}{\leftarrow} s_j$
      else
        $S_{A_{\text{in}}} \overset{\text{put}}{\leftarrow}$ copy of s_j, sequence $(*)_A$
      end if
    else if all $s_j \in S_{\text{out},i}$ **are** SINGLE **then**
      $I \leftarrow$ item type of all $s_j \in S_{\text{out},i}$
      $S_{A_{\text{in}}} \overset{\text{put}}{\leftarrow}$ new SINGLE $(I)_{A_{\text{in}}}$
    else if all $s_j \in S_{\text{out},i}$ **are** MULTI and **all** $T(s_j)$ **are** equal **then**
      $T \leftarrow$ sequence tuple $T(s_j)$
      $n \leftarrow$ number of sequences in $S_{\text{out},i}$
      $S_{A_{\text{in}}} \overset{\text{put}}{\leftarrow}$ new MULTI $(T)^n_{A_{\text{in}}}$
    else
      $S_{A_{\text{in}}} \overset{\text{put}}{\leftarrow}$ new SPLIT $< S_{\text{out}} >_{A_{\text{in}}}$
    end if
  end for
end function

function INHERITMERGE(Merge G)
  Outgoing section $B_{\text{out}} \leftarrow \text{Start}(B_{\text{out}}) = G$
  Incoming sections $N_{\text{in}} \leftarrow \{ A \in N \mid \text{Goal}(A) = G \}$
  Outgoing sequences $S_{\text{out}} \leftarrow S_{B_{\text{out}}}$
  for $S_{\text{out},i} \subseteq S_{\text{out}}$ **do** ▷ Independent subsets using Equation 2.1.
    for $s_j \in S_{\text{out},i}$ **do**
      $N_{\text{in,exclusive}} \leftarrow$ sections s_j is exclusive to ▷ Using the definition in Section 2.2.4.
      $N_{\text{in,shared}} \leftarrow N_{\text{in}} \setminus N_{\text{in,exclusive}}$
      for $A_{\text{in}} \in N_{\text{in,exclusive}}$ **do**
        $I_{\text{inherit}} \leftarrow I_M(s_j) \cap I_M(A_{\text{in}})$
        if I_{inherit} has one unique item type **then**
          $S_{A_{\text{in}}} \overset{\text{put}}{\leftarrow}$ new SINGLE $(I_{\text{inherit}})_{A_{\text{in}}}$
        else if s_j **is** MULTI **then**
          $T_{\text{inherit}} \leftarrow$ sequence of s_j, filtered to item types in I_{inherit}
          $S_{A_{\text{in}}} \overset{\text{put}}{\leftarrow}$ new MULTI $(T_{\text{inherit}})_{A_{\text{in}}}$
        else
          $S_{A_{\text{in}}} \overset{\text{put}}{\leftarrow}$ copy of s_j, sequence $(*)_A$
        end if
      end for
      if $N_{\text{in,shared}}$ **is not** empty **then**
        s_j is converted to a new merge sequence, s^*_j
        All $A \in N_{\text{in,shared}} \overset{\text{put}}{\leftarrow} s^*_j$ ▷ Note that all sections inherit the same sequence.
      end if
    end for
  end for
end function

2.3 Design Considerations

While the possible throughput of a solution is not directly subject to optimization, it typically has to meet minimal requirements to be viable. This section analyzes the dynamic properties of the movers and estimates the throughput for the primitives presented in Section 2.2.2 under considerations of the model defined in Section 2.1. As discussed in Section 1.3, the presented problem typically gives rise to solutions with a higher density of movers over the available space compared to most environments for autonomous vehicles. One reason for this is the relatively small size difference between mover and field. Despite the whole area of the field being available for navigation, trajectories rarely involve straight lanes longer than an order of magnitude larger than the mover length. Additionally, the fixed station positions, contrary to the more distributed trajectory starts and goals of pick-and-place systems, permit a static lane layout, while the expected throughput of the system requires optimized flow of densely stacked movers.

All numeric values in this section assume square $118\,\mathrm{mm} \times 118\,\mathrm{mm}$ movers, representing the smallest mover size currently offered by Planar Motor [46]. Including tolerances to avoid collisions, a square primitive edge length of $l_P = 120\,\mathrm{mm}$ is a reasonable base size for the following calculations. While larger movers generally result in lower flow rates, the resulting considerations are valid for other sizes and non-square movers as well.

2.3.1 Mover Dynamics

The index length l is defined as the smallest possible distance between two movers traveling in the same direction along a straight lane without colliding. This distance is dependent on the movement direction and relative rotations of the movers. This section presents the simplest case, in which both movers are oriented along the movement direction. Then, the index length is equal to the primitive edge length, $l = l_P = 120\,\mathrm{mm}$. Figure 2.11 visualizes some exemplary index lengths. If the mover orientation is not normal to the movement direction or the lane length is not an integer multiple of the index length, sections with larger index lengths occur.

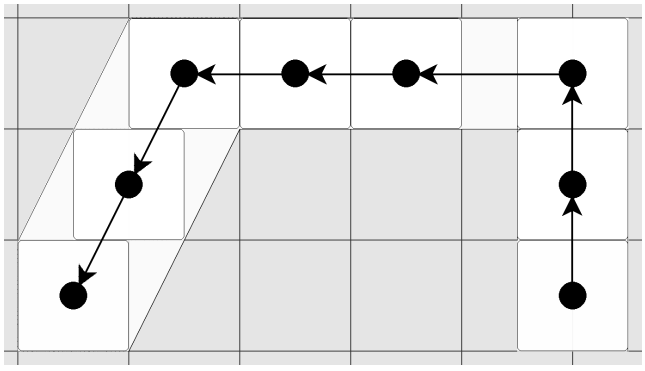


Figure 2.11 Straight lanes connected by corners, with marked waiting positions. The index length is dependent on the lane orientation and overall length.

Traveling at a constant maximum velocity $v_{\max}$, a mover can then traverse l in the minimal index duration

$$\tau_v = l / v_{\max}$$

If a mover is part of a queue, it may only be able to shuffle one index length before coming to a halt again. This can be represented as point-to-point movement, in which a stationary mover accelerates until reaching the half-way point before decelerating symmetrically and coming to a halt at the new index position. Then, the index duration is

$$\tau_{\mathrm{P2P}} = 2\tau_{a,l/2}$$

with $\tau_{a,d}$ being the time required to traverse a distance d starting from a standstill and, symmetrically, breaking to a standstill.

As defined in Section 2.1, the mover is restricted by a maximum velocity $v_{\max}$ and acceleration $a_{\max}$. If $v_{\max}$ is reached before the distance d, the mover cannot accelerate further and the velocity profile plateaus. For a mover with the parameters defined above, this results in a duration of

$$\tau_{a,\max} = \frac{v_{\max}}{a_{\max}} = 100\,\mathrm{ms}$$

and distance of

$$d_{a,\max} = \frac{1}{2} a_{\max} \tau_a^2 = 100\,\mathrm{mm}$$

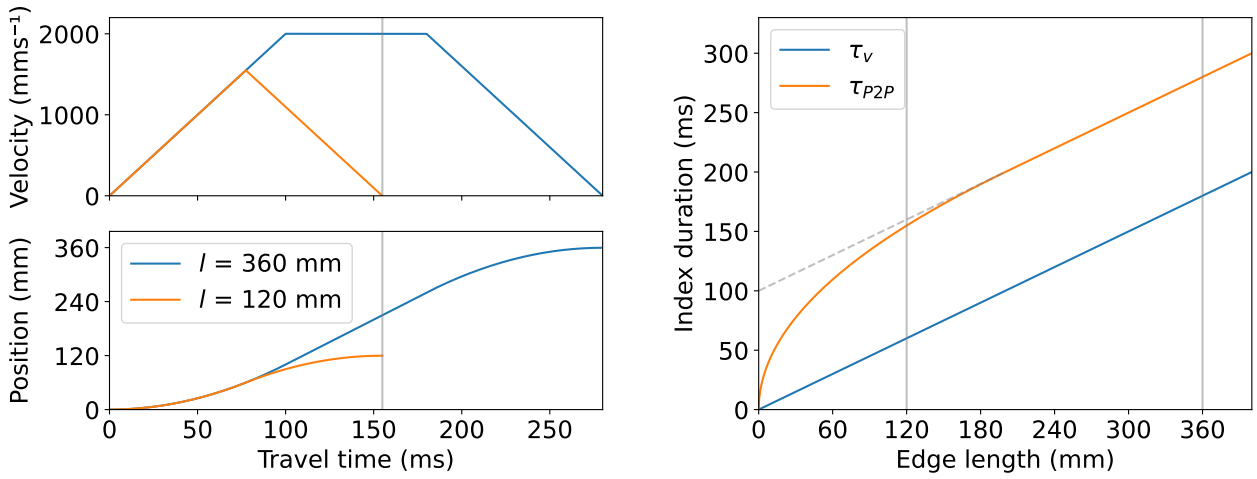
to accelerate from 0 to $v_{\max}$. Specifically, this results in distance traversal times of

$$\tau_{a,l/2} = \begin{cases} \sqrt{\frac{l}{a}} & \text{for} \quad d_{a,\max} \geq \frac{l}{2} \quad \text{(acceleration limited)} \\ \frac{1}{2}(\tau_{a,\max} + \tau_v) & \text{otherwise} \qquad \text{(velocity limited)} \end{cases}$$

and

$$\tau_{a,l} = \begin{cases} \sqrt{2\frac{l}{a}} & \text{for} \quad d_{a,\max} \geq l \quad \text{(acceleration limited)} \\ \frac{1}{2}\tau_{a,\max} + \tau_v & \text{otherwise} \qquad \text{(velocity limited)} \end{cases}$$

As shown in Figure 2.12a, it follows that all movers up to an edge length of $200\,\mathrm{mm}$ are acceleration limited during point-to-point movement, while all larger movers are velocity limited. For $l = 120\,\mathrm{mm}$, the mover arrives after $\tau_v = 60\,\mathrm{ms}$ and $\tau_{\mathrm{P2P}} = 155\,\mathrm{ms}$, for $l = 360\,\mathrm{mm}$ after $\tau_v = 180\,\mathrm{ms}$ and $\tau_{\mathrm{P2P}} = 280\,\mathrm{ms}$. Despite this, as shown in Figure 2.12b, for typical mover dimensions above $l_G = 120\,\mathrm{mm}$ both durations are approximately linear with respect to the edge length.



(a) Velocity and position profiles of point-to-point movements.

(b) Index durations for different mover edge lengths.

Figure 2.12 Velocity, position profiles and index durations for different index lengths l.

2.3.2 Primitive Flow Rates

A flow F is defined as the number of movers that can traverse any part of the network per time. Equivalently, the traversal time $t = 1/F$ is the minimal time two movers must be separated to avoid collision. For this general analysis, one can distinguish between two types of flow. The optimal flow F_O and time t_O represent the largest possible flow through any primitive. This usually requires all movers to enter the primitive at a set time and velocity and exit unobstructed. Alternatively, the shuffling flow F_S and time t_S represent movers moving point-to-point along all index positions, coming to a halt on all awaitable nodes. A typical effective maximal flow lies between these two estimates.

Straight Lanes

For unidirectional travel, the movers are exactly the index length l apart and move synchronously in the same direction. This results in the traversal times of

$$t_O = \tau_v$$
$$t_S = \tau_{\mathrm{P2P}}$$

For the mover parameters defined above, this results in flow rates of $F_O = 1000\,\mathrm{min}^{-1}$ and $F_S = 387\,\mathrm{min}^{-1}$ respectively. If bidirectional travel is required, a switch in directions requires the section to be completely cleared, adding a significant flow penalty to any such lane.

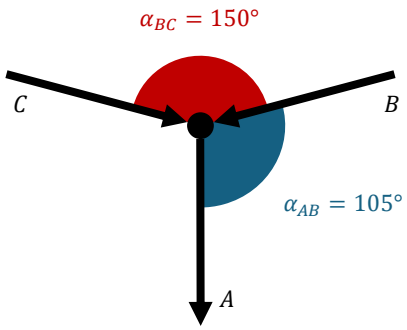


Figure 2.13 Notation for angles between lanes. Decisive is the angle along the connecting node, which may differ from the angle between movement direction vectors.

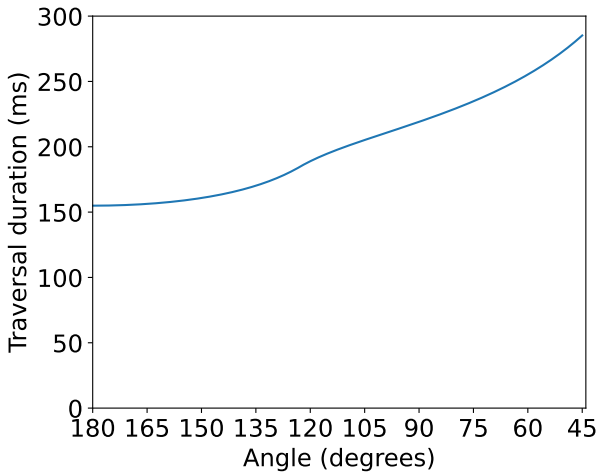


Figure 2.14 Optimal corner traversal duration t_O for different angles between connected lanes. $180°$ represents the limit for parallel lanes.

Corners

As discussed in Section 2.1, all direction changes require the mover to come to a complete stop before continuing. For corners with nearly parallel lanes, the traversal times approach the minimum

$$t_O = t_S = \tau_{\mathrm{P2P}}$$

while for $90°$ angles, the traversal times increase to

$$t_O = 2\tau_{a,l}$$
$$t_S = 2\tau_{\mathrm{P2P}}$$

as the incoming mover has to await the outgoing mover clearing sufficient space before being able to enter it. Following the notation of Figure 2.13, this overlap and consequently the traversal times increases for smaller angles as shown in Figure 2.14. Below $135°$, the incoming mover enters the corner footprint from the side, requiring the outgoing mover to clear a part of the footprint completely before the incoming mover can enter it. Due to the rectangular shape of the movers, the collision distance is not constant over the corner angle. The resulting flow rates of $F_O = F_S = 387\,\mathrm{min}^{-1}$ for nearly parallel lanes and $F_O = 273\,\mathrm{min}^{-1}$ and $F_S = 194\,\mathrm{min}^{-1}$ for $90°$ angles are smaller than for straight lanes, limiting the possible throughput of the overall section.

Intersections

Intersections are required for two lanes to cross safely. The intersection area of the two lanes is exclusive to the active lane, movers on the blocked lane may not enter it. The active lane then forms a straight path with corresponding flow. If the incoming and outgoing movement directions are not parallel, the intersection behaves similar to a corner.

When switching directions, the blocked mover must wait for the previous mover to clear the intersection before entering it. This resulting overhead for the traversal time is dependent on the geometry of the intersection. For two lanes intersecting at a right angle and square movers, the traversal times for the first mover after switching are

$$t_O = 2\tau_v$$
$$t_S = 2\tau_{a,l}$$

The smallest flow occurs if directions are alternated after each mover. Then $F_O = 250\,\mathrm{min}^{-1}$ and $F_S = 136\,\mathrm{min}^{-1}$ per lane.

Splits and Merges

Splits and merges allow multiple paths to be combined, reducing the overall space requirements and number of intersections. Merging paths results in their flows summing up, increasing the requirements on the following parts of the network.

In this work, only two-way splits and merges are considered. While other combinations may be more space efficient, increasing the number of connected lanes consequently decreases the angles between them. Similar to the limitations of corner angles discussed above, reducing the angle between two lanes results in a larger overlapping area that must be cleared before another direction can be activated.

Stations

As discussed in Section 2.1, interaction stations are modeled as single positions on which movers must remain for the interaction duration τ_I before leaving. This delay places an upper limit on the achievable flow, which is further reduced by the time required to replace the movers between interactions. Therefore, the incoming and outgoing lanes of the station should ideally be placed on opposite sides to allow a simultaneous movement of the outgoing and incoming mover. Such an arrangement then results in traversal times of

$$t_O = t_S = \tau_{\mathrm{P2P}} + \tau_I$$

For $\tau_I = 100\,\mathrm{ms}$, this allows the station to provide a flow of $F_O = F_S = 235\,\mathrm{min}^{-1}$. In typical applications τ_I may be much higher, resulting in stations having the lowest flow of all primitives of the network. Despite this, flows of multiple stations are often merged onto a single section, requiring all elements of the network to be considered to determine the overall possible flow.

2.4 Design Strategies

This section summarizes the findings of this chapter and formulates a number of design strategies to provide a guideline for the proposed algorithm most likely leading to efficient and reliable solutions.

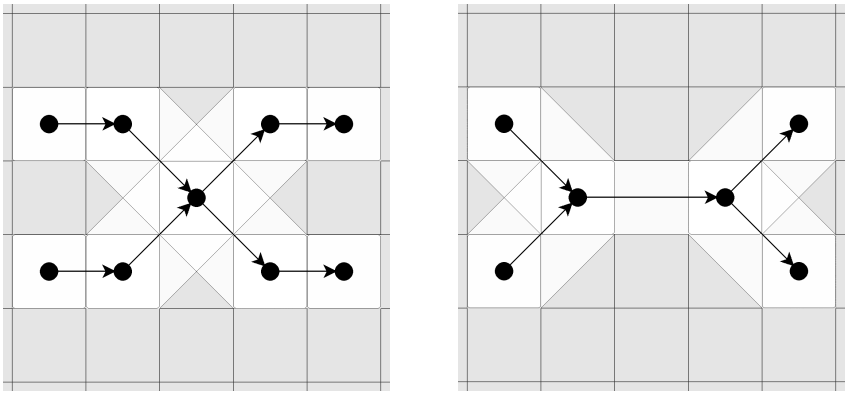
The static design of the item sequences allows the algorithm to predict the expected item flow. This knowledge can be used to lay out necessary paths and ensure sufficient capacity along all parts of the network. Making the expected flow available to the online controller allows the use of a more proactive strategy, simplifying the reliable fulfillment of the receiver station sequences.

The system must have sufficient capacities to account for variances in interaction duration. For small errors, this can be achieved by integrating waiting queues and additional movers. Larger errors, such as the loss of a station, must be accounted for by adding additional redundant paths or preparing alternative, reduced solution graphs. The employed real-time controller must then be able to detect or be notified of the error and have the ability to dynamically compensate.

As discussed in Section 2.3.2, intersections and corners have reduced flow compared to straight lanes. For high flow sections, corners and intersections should therefore be avoided wherever possible. If the primitives are required, the angle between any two joining edges should be maximized to minimize the resulting overhead. A valid network therefore requires a minimum of 90° separation as a good compromise between flow and flexibility, as this retains the ability to form rectangular grid patterns. This ability is crucial for achieving densely packed parallel lanes as can be seen in the networks of Figures 2.15c and 2.15d.

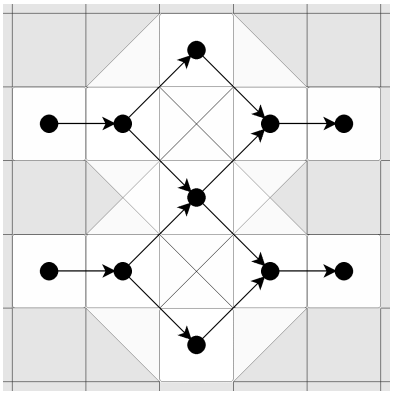
Furthermore, stations are expected to have the lowest flow of all primitives. For this reason, station interfaces serve as a buffer and staging point for their attached station, allowing for an optimal exchange of movers by compensating for any variance in the interaction duration. To achieve this, they must be placed as close as possible to the station, while the sink and source interface should be placed as far apart as possible, if possible on opposing sides of the station along the field border. Each station requires at least two interfaces for operation, one sink and one source. Following the proposed 90° separation, stations with sufficient space have the option to use a third interface. In this case, the third interface is used as either a second sink or source and serves as an alternative connection to the station. While using this option generates a corner within the station and consequently reduces the achievable flow, environments with little available space may benefit from the increased flexibility. Additionally, using three interfaces simplifies the controller design by avoiding mixing flows of different paths along a section. Another conclusion from this is that using a single path between any two stations per direction will not limit the possible throughput, while reducing the required space and simplifying the network layout.

Enforcing the 90° separation on cross primitives results in a rigid arrangement, severely limiting the flexibility of the surrounding graph. This can be alleviated by replacing the cross primitive with a merge-arc-split arrangement, effectively stretching the intersection along one direction as shown in Figure 2.15b. Both types of intersections have strengths and weaknesses, which should be considered when planning the network. While crosses have a smaller minimal footprint than merge-splits, the latter has the ability to directly conform to the surrounding network without the need for additional corners, frequently resulting in lower overall space requirements as can be seen in Figures 2.15a and 2.15b. Additionally, merge-splits do not need additional bypasses to allow for same-side connectivity. Whereas Figure 2.15b can route movers between the top input and output, using a cross requires an arrangement as shown in Figure 2.15c, significantly increasing the size of the structure. This ability however leads to an increased sequence complexity of a merge-split design if no same-side connectivity is required. In a network with multiple chained intersections, the rules of inheritance presented in Section 2.2.4 may result in merge sequences inheriting split sequences already containing merge sequences from the following intersection. If these dependencies cannot be separated by single or multi sequences, this will result in long dependency chains throughout multiple intersections, reducing the efficiency of the system. On the other hand, crosses without bypasses are completely passed over by sequences.

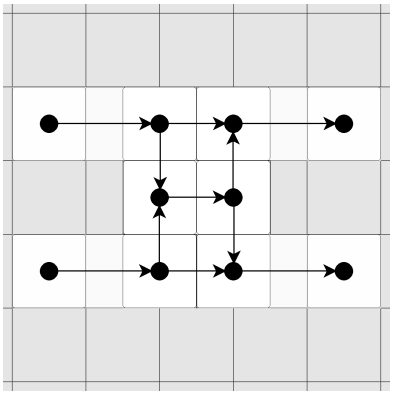


(a) Intersection using Cross

(b) Intersection using Merge and Split



(c) Intersection with Bypass using Cross



(d) Intersection with Bypass using Merge and Split

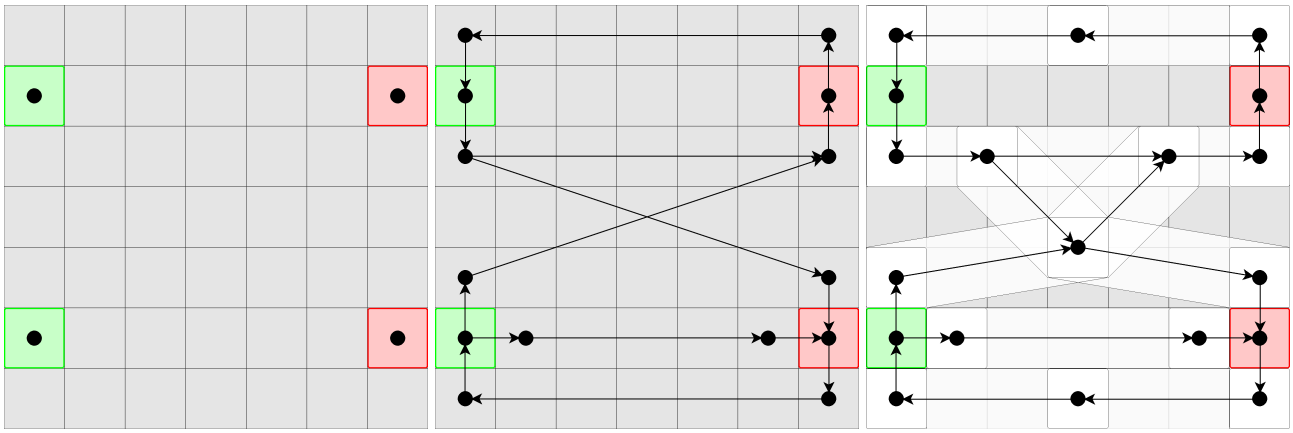
Figure 2.15 Examples of different primitive arrangements for intersecting two lanes. The outermost primitives are fixed in place and all joining edges must be separated by at least $90°$.

Comparing the expected performance between the intersection variants discussed in Section 2.3.2 shows that crosses have a larger flow for pure opposing-side routing, especially when the flow is biased along one direction or a batch-wise routing is possible. While the throughput for Figures 2.15a and 2.15b are comparable, the arrangement in Figure 2.15d results in chained $90°$ corners with large overlaps, reducing the opposing-side flow. While the presented merge-splits are invariant to the number of direction changes, even when alternating directions the traversal time for the network in Figure 2.15d with $t_S = 2\tau_{\mathrm{P2P}}$ is larger than for the network in Figure 2.15c with $t_S = 2\tau_{a,l}$. Despite this, adding bypasses to a split-merge as shown in Figure 2.15d may be beneficial by increasing the potential flow for same-side routes. In the shown arrangement, the merge-split bypasses are straight lanes allowing for an uninterrupted flow, while bypassed crosses in the arrangement of Figure 2.15c require a $90°$ corner along their bypasses. Additionally, bypassed merge-splits tend to fit better into rectangular grid patterns than crosses, simplifying the placement in tight arrangements. An advantage of the bypassed cross arrangement is the ability of movers to enter the bypass while the opposing direction is active, allowing a second mover to await the intersection. After switching directions both movers may then proceed simultaneously, reducing the overall overhead of the switch.

In summary, merge-splits tend to perform better in tight spaces and for networks with high throughput along bypasses, while providing same-side connectivity even without bypasses. On the other hand, crosses offer reduced control complexity and better opposing-side throughput. They perform best on large fields with the ability to queue and batch-process movers and sufficient space allowing the rigid arrangement to be integrated without the use of corners.

3 Concepts of the Offline Preprocessing

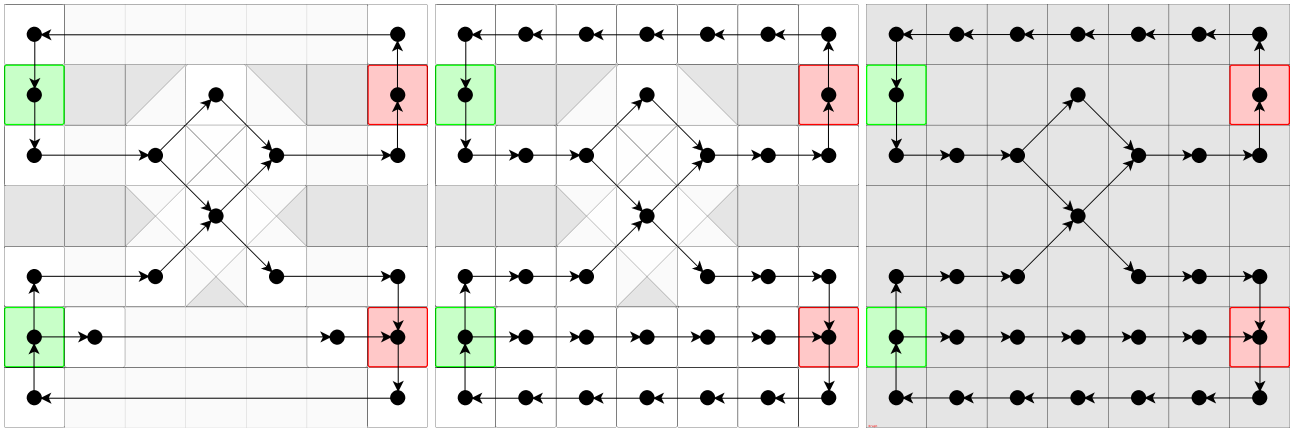
The contribution of this work is split into two parts, the offline solution generation on an external computer and the online control performed on a physical or simulated real-time controller. This chapter presents the offline preprocessing algorithm, which consists of a pipeline with four separate steps: Flow generation, network initialization, network optimization, and post-processing. The input to the pipeline is an environment defined in Section 2.2.1, encoding the problem statement as visualized in Figure 3.1a. The output of the pipeline is a graph as defined in Section 2.2.3 and shown in Figure 3.1f, containing a valid solution to be used by the controller. All data generated and propagated in the pipeline is contained in networks defined in Section 2.2.2.



(a) Initial field, containing only the stations specified in the environment.

(b) Output of the flow generation. Each expected flow is represented by a path through the assigned interface.

(c) Network after initialization. All crosses, splits and merges are inserted. Corners are inserted to split long arcs.



(d) Optimized network, having shifted non-fixed primitives and added or removed corners where necessary.

(e) Output of the post-processing, including wait primitives along long arcs.

(f) Output graph of the offline solution generation, ready to be used by a controller.

Figure 3.1 Overview of the offline preprocessing pipeline, generating a viable solution from an environment.

At a high level, the generation of a viable solution can be further divided into two separate problems, following the approach proposed by Peddada et al. [40] elaborated in Section 1.2. Initially, one must determine which paths, representing connections between interaction stations, are required to balance the expected item flow between stations. The presented flow generation algorithm solves this by generating the required station interfaces and determining the best arrangement of paths between source and sink interfaces as shown in Figure 3.1b. During this, a number of metrics need to be balanced, most notably the necessary redundancy of the paths, the limited available space and the estimated solution complexity and performance. While an informed decision on these cannot be made without knowledge of the layout of the field and stations, an abstraction is used to simplify the process.

The second problem concerns the physical routing of the determined paths. As discussed in Section 2.2.2, a number of conditions for the connectivity and positions of the components in a network need to be fulfilled for the network to be valid. As the second step of the pipeline, the network generation expands on the flow paths to create a network fulfilling the connectivity requirements, ensuring that no two arcs intersect and that all primitives have the required number of incoming and outgoing arcs. Initially, for each path in the generated flow, an arc is generated. Then, at each intersection of two of these arcs an intersection structure is inserted into the network. As discussed in Section 2.4, this can either be a cross primitive or a merge-split arrangement, dependent on the environment requirements. While a cross generally performs better in large networks, a merge-split usually requires less overall space. Afterwards, if any interface is connected to more than one other station, Split and Merge primitives are inserted until the primitive connectivity requirements are fulfilled. Finally, any remaining arc above a given length is split into multiple shorter arcs by inserting corner primitives, aiding in the following optimization.

The network generated using this method likely has multiple violations such as overlapping primitives or small arc angles to be considered valid as is the case in Figure 3.1c. The network optimization step of the pipeline addresses this by using a force-based iterative optimization to converge to a valid solution as shown in Figure 3.1d. For this, all non-fixed primitives are shifted, following forces related to the existing violations. If an equilibrium of forces is reached that does not satisfy all validity criteria, corner primitives are added or removed to allow further progress towards a solution. After a sufficient solution is found, a number of post-processing steps are performed to generate a finalized network as shown in Figure 3.1e. This involves a coarsening of the primitive positions and the generation of waiting primitives along qualifying arcs before the final network is converted to the output graph shown in Figure 3.1f.

If the autonomously operating pipeline is unable to generate a valid solution for an environment, a supervised operation is possible. This allows the expert using the system to intervene, modifying the generated network before resuming processing at any point of the process. The compact data structures of networks and graphs are suited for an export and import into human-readable and -modifiable formats such as JSON or a graphical user interface allowing manipulation of the structures. Furthermore, the force-based placement optimization is well-suited for supervised operation, as all primitives are moved iteratively on the field. In addition, a checkpoint of the network may be exported or imported at any iteration, while a network can be intuitively visualized by drawing the two-dimensional field with the placed primitives. This allows for a live visualization of the current network during runtime or the generation of a video file for later analysis. Additionally, the forces acting on the primitives can be visually represented by component-wise vectors, allowing for an intuitive overview of the governing forces of the system. An expert has various options to modify the network, such as moving, fixing, adding, removing or reordering primitives to guide the algorithm towards a desired solution.

3.1 Flow Generation

The first step of the offline pipeline is the flow generation, which determines the paths necessary to fulfil the requirements of the system. As discussed in Section 2.4, the design of the problem statement allows these paths to be defined statically, based on the expected product flow and required redundancy between the stations. Each generated path connects interfaces of two stations and represents all item types flowing from the source to the sink station. Consequently only one path exists per pair of station and direction. Paths connecting provider to receiver stations are traversed by full movers, while paths from receivers to providers are used by empty, returning movers. The paths and interface assignment are determined in three separate steps. First, the required paths between stations are defined. Then, considering all possible locations and combinations, the required interfaces are generated and finally matched with the paths.

3.1.1 Station Flow Generation

For generating the required paths without redundancy, it is sufficient to fulfill each individual station requirement by balancing the flow of the system. This is accomplished if, for every station, the expected incoming flow matches the outgoing flow and the environment flow specifications. The combined outgoing paths of each provider must match the specified flow for each item type and the combined flow of incoming empty movers, and vice versa for receivers. From this follows that, if the overall system is not balanced, no exact solution exists and the environment is not well-balanced.

The presented algorithm uses a greedy, minimal distance approach. This promotes local connectivity with the intention to reduce the number of required intersections, which limit the possible flow through the system as discussed previously. For each item type, if only one provider or receiver exists, the assignment is forced and the corresponding paths are created. For all remaining item types, each receiver is matched with providers, sorted by ascending euclidean distance between the stations, until its demand is saturated. If redundant paths are required, this algorithm may be modified to generate matchings until a given threshold above the required flow can be provided. In the extreme case, all providers have paths to all receivers and vice versa. This however significantly increases the complexity of the network and should be avoided if possible.

Alternative approaches minimize the number of paths or total path length, arguing that this results in the least space required on the field, therefore simplifying the placement. While this could result in improved solutions for larger environments, for most practical applications these more complex approaches yield the same or similar results to the one presented above.

3.1.2 Interface Generation

Following the established strategies, interfaces should be placed on opposing sides of the station wherever possible. With all stations placed along a border, the proposed algorithm therefore initially attempts to generate the interfaces along the border of the field adjacent to the station. Consequently, the third interface can be placed at the side of the station facing towards the field center. It is possible that one of these spaces is blocked, either by the field border or another station or interface. In this case, the other two positions must be populated, as each station requires both a sink and source interface. Consequently, if only a single space is available for all arrangements of interfaces, the environment cannot be solved. As station interfaces cannot overlap if two stations are too close, the section between them may only contain the interface of one of the stations. In this case, assuming both stations can generate at least two interfaces, the contested interface is assigned to the station with the smaller average distance to all other stations. This metric attempts to anticipate the available space on the field close to the station. A station with three possible spaces has increased flexibility, either choosing two preferred locations or making use of all three, which serves as an integrated split or merge primitive which does not have to be accommodated for next to the station.

3.1.3 Interface Flow Assignment

After determining the required flows between stations and the available interfaces, each station flow must be matched with a sink and a source interface. As they are unidirectional, each interface must be connected to either only incoming or only outgoing paths. This matching is chosen to minimize the number of intersections of paths, simplifying the design and avoiding the low flow along intersections as discussed in Section 2.4.

The proposed algorithm approaches this by using geometric intersections and exhaustive testing of all combinations. Initially, for all flows generated between stations, each interface of the source station is connected to each interface of the sink station by a straight line path. Next, an intersection map containing all pairs of paths is generated. Due to the requirement of the model in Section 2.1, that all stations are placed at the border of a convex field, determining whether two straight paths intersect is equivalent to determining whether any two continuous paths between the same interfaces intersect. For general fields, determining an accurate number of intersections requires a more extensive routing and collision detection approach. A matching then consists of a set of active paths such that each station flow is satisfied and all active interfaces are conflict-free, having either incoming or outgoing active paths. For each matching, the number of intersecting active paths can then be computed using the intersection map.

While the number of valid matchings is non-polynomial with respect to the number of interfaces and flows, testing all combinations is still feasible for typical environments and guarantees finding the optimal matching of the metrics. The number of matching for a set of interfaces and flows can be approximated by considering the possible combinations for all stations separately before combining them for the overall system. Then, the number of matchings N is

$$N = \prod_{i=1}^{n} N_S(i)$$

for n stations with N_S combinations of interface assignments each. Each station has a fixed number of incoming and outgoing paths to be assigned, while each interface can either serve as a sink or source for the corresponding paths. From this follows that for stations with two interfaces, as one must be a sink and the other a source or vice versa, $N_S = 2$. For stations with three interfaces more possible combinations exist as designating two of the interfaces as sink or source allows the corresponding paths to be assigned in any combination to either interface. This results in

$$N_S = 3 \cdot 2^{n_I} + 3 \cdot 2^{n_O} = \mathcal{O}(2^{n_I} + 2^{n_O})$$

combinations for n_I incoming and n_O outgoing paths. Consequently, the total number of matchings is

$$N = (6 \cdot 2^{n_p})^{n_3} \cdot 2^{n_2} = \mathcal{O}(2^{n_3 n_p + n_2})$$

for n_3 three-interface stations with n_p connecting paths each and n_2 two-interface stations, assuming $n_I = n_O = p$ and equally distributed paths.

For larger systems, heuristic methods such as simulated annealing [47] can be used to find a feasible, but not guaranteed optimal, solution in constant time. An alternative approach to reduce the number of potential matchings is to force the use of paths without intersections, such as those following the field border. While these paths are likely contained in the optimal solution, they are not guaranteed, resulting in this method to not be guaranteed optimal either.

A third method, reducing the search space while ensuring an optimal solution, is a topologically consistent reduction. For the global layout, the relative arrangement of three-interface stations is only relevant if the lone interface type is in the center position as shown in Figure 3.2b. Otherwise, the same-type center and side interfaces will result in the same number of global intersections as shown in Figure 3.2a. This leads to

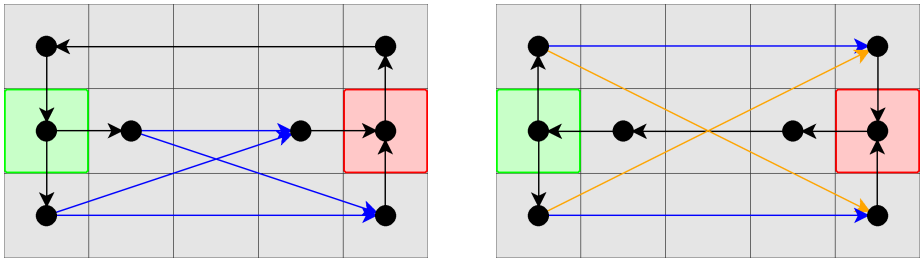
$$N_S = 2^{n_I} + 2^{n_O} - 2$$

and

$$N_C = (2 \cdot 2^{n_f} - 2)^{n_3} \cdot 2^{n_2}$$

with an approximate reduction of the search space of factor 3^{n_3}. While still resulting in a non-polynomial complexity, this approach matches the strategy proposed in Section 2.4 to only make use of the two outer interfaces. If stations are to make use of all three interfaces, a second search retaining the optimal assignments of all interface types will result in an equivalent solution to a search over the full search space while requiring the computations of less combinations overall. Then, the second search can be used to minimize the number of interfaces serving multiple paths, as interfaces serving a single path do not require an external split or merge and reduce the complexity and required space on the field close to the station.

Solely using minimal number of intersections heuristic will result in multiple equivalent optimal matchings for most environments. In this case, the proposed algorithm optimizes the matchings based on the overall number of occurrences of the contained active paths in all optimal matchings. For this, each path is assigned a score consisting of the number of optimal matchings in which it is active. The best matching then has the highest sum of scores of all its active paths, assuming that paths frequently found in optimal solutions are preferable in general. Additionally, this approach has shown to reliably retain symmetries from the environment to the matching. If all station flows are bidirectional, with each station either having no flow or sending and receiving items from any other station, the optimal matching is symmetric as well and can be reversed by flipping the directions of all paths. As all following steps of the proposed offline pipeline are symmetrical with regard to the movement direction, this property can be considered during post-processing if one of the options proves superior. One example where this may be the case is if one of the solutions has more queue capacity after a critical section than before it, in which case flipping the directions might improve the robustness of the solution.



(a) Lone interface at the side. Any of the blue paths will result in the same number of intersections.

(b) Lone interface in the center. The blue and orange paths will result in a different number of intersections.

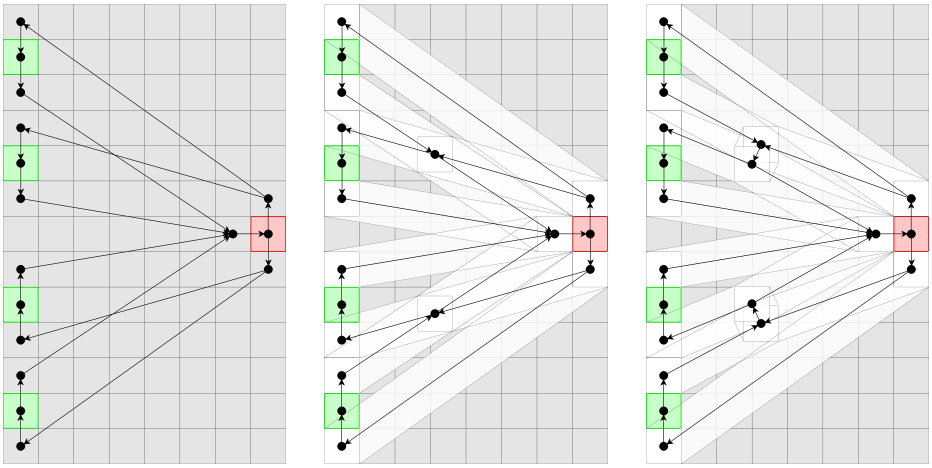
Figure 3.2 Topologically equivalent reduction of station interface matchings. If the lone interface is at the side, the other assignments are equivalent.

3.2 Network Initialization and Transformation

The implementation of the networks closely follows the abstract description of Section 2.2.2. However, in order to provide an efficient datastructure for the presented algorithms, it is essential that all properties and relationships between the different components are easily accessible and synchronized. This section elaborates on the implemented operations for the generation and modification of the network, as well as the efficient computation of geometric properties such as the overlap detection.

The initial network of the pipeline is generated after the flow assignment, based on the chosen interfaces and connecting paths. As a first step, station and interface primitives are generated in their designated positions. For each generated path, an arc is generated between the interfaces. Each path then contains three arcs, starting from the station to its interface, cross the field along the new arc to the other stations interface, and ending at the destination station. From this point on, all modifications of a contained arc or primitive must also update the path. The generation of paths at this stage is necessary, as they contain the routing information between stations and are uniquely matched to the set of items and corresponding flows, which must be referenced later in the pipeline during sequence generation.

After the path generation, if any intersections exist in the matching of the flow assignment, the corresponding arcs are split using either a cross or a merge-arc-split arrangement as shown in Figure 3.3. Crosses can be placed at the computed intersection point, while the arc connecting the merge and split requires an offset of the primitives, placed parallel to the bisector of the initial arc directions. It must be ensured that, if intersections are located close together, no new arc intersections are created by this arrangement. If this is the case, the connecting arc length must be reduced or an alternative interface matching must be found. Alternatively, the assignment of interfaces contested by multiple stations mentioned in Section 3.1.2 can be modified.



(a) Interface and path matching of the flow generation.

(b) Intersection resolution using cross primitives.

(c) Intersection resolution using merge-split arrangements.

Figure 3.3 Example of different options for resolving intersections in the generated flow matching, with the minimal amount of two required intersections for the shown connectivity.

As stated in Section 2.2.2, interfaces may only connect to one other primitive in addition to their station. If any interface serves two or more paths, this requirement is violated and additional split or merge primitives must be inserted. Consequently, for an interface serving n paths, $n - 1$ primitives are arranged in a binary tree structure rooted in the interface. The exact structure of the tree is subject to optimization and dependent on the overall network and arrangement on the field. One arrangement is a balanced tree as shown in Figure 3.4, which typically provides a symmetric solution for a symmetric environment and quickly distributes the flow among the branches. Another arrangement, implemented in the presented algorithm, instead generates a simpler, unbalanced tree as shown in Figure 3.5. For this, the connected arcs are sorted by euclidean length and the required split or merge primitives are placed at equidistant points along the longest arc. Then, each primitive is connected to the primitive matching its sorted position. This ensures that no new intersections are generated and typically results in an asymmetric arrangement with each path feeding into or from a central spine.

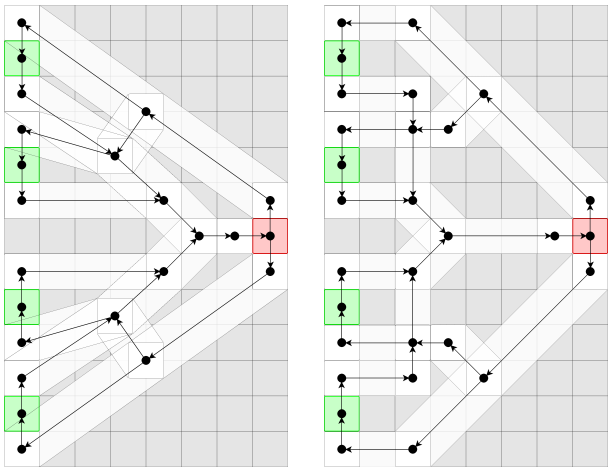


Figure 3.4 Initial and optimized networks using balanced trees for interface resolution.

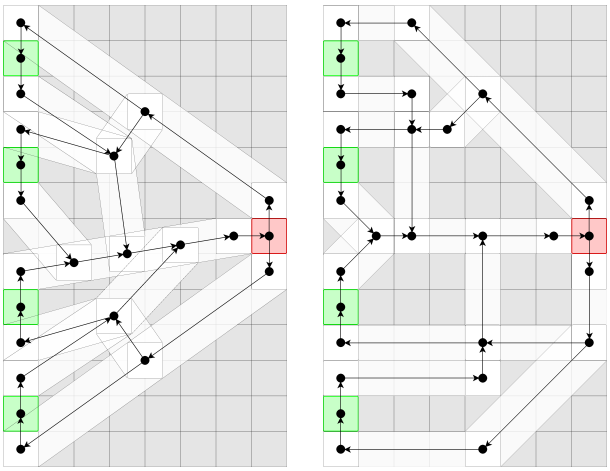


Figure 3.5 Initial and optimized networks using unbalanced trees for interface resolution.

The optimization algorithm presented in Section 3.3 has the ability to modify the network by adding or removing corners. Arcs are by definition straight connections between two primitives and are therefore split by a corner if they fail to clear some other feature without buckling. During the insertion, the new corner is placed in the center of the arcs tail and head position. Then, the old arc is updated to originate from the corner, while a new arc is generated to connect the old tail to the corner. Furthermore, all paths containing the original arcs must be updated to include the new arc in the correct position of the arc chain. A removal of a corner reverts these changes by mapping the outgoing arc to originate from the incoming arcs tail and removing the incoming arc from the path and network. The same procedure is applied to wait primitives during post-processing, although they must not be shifted afterwards to maintain a parallel incoming and outgoing arc.

As a final modification to the initial network, arcs above a certain length are split into sections by inserting corner primitives at equidistant points. This has shown to improve the robustness of the optimization, as long arcs running diagonally across the field tend to prevent expansion into the cut off areas by behaving similar to field borders. Inserting one or more corners during initialization allows the placement optimization to buckle the arc towards the field borders from the outset instead of requiring the network modification to insert a corner. If the corner is not required in the solution, the connected arcs will remain parallel and can be re-combined later without affecting the network.

During the iterative optimization loop, a number of checks must be performed repeatedly between nearby arcs and primitives. As the number of computations are quadratic to the number of primitives, it is essential to efficiently compute these properties. Namely, these computations are the current and collision distance between all nearby primitives, the same distances between primitives and the centerline of nearby arcs, and the relative angles between arcs connected to the same primitive. Computation of the relative arc angles follows the established definitions, as visualized in Figure 2.13.

To reduce the number of required checks for the distance calculations, an upper limit $d_{\max}$ is defined above which no interaction occurs. The proposed algorithm uses

$$d_{\max} = d_{\mathrm{col,max}} f_o \quad \text{with} \quad d_{\mathrm{col,max}} = \|l_P\|$$

based on the collision distance d_{col} and $f_o \geq 1$, a factor accounting for the perimeter outline discussed in the next section. $d_{\mathrm{col,max}}$ is the largest possible collision distance, based on the diagonal length of the rectangular primitive footprints with axis-aligned edge lengths $l_P = (l_{P,1}, l_{P,2})^T$.

A collision grid similar to the grid-square algorithm proposed by Fruchterman and Reingold [43], is employed, allowing for an efficient lookup of all potential collisions. The grid spans the complete field and consists of buckets with edge lengths of $d_{\max}$. After each modification of the network or primitive positions, all buckets are filled with the primitives currently located in the corresponding area of the field. The chosen edge length then guarantees that, for any primitive on the field, only primitives contained in the same or one of the eight adjacent buckets can be closer than $d_{\max}$. Consequently, an arc can only interact with primitives that are contained or adjacent to a bucket passed by the mover traveling along its centerline. Employing a voxel traversal algorithm, such as the one proposed by Amanatides and Woo [48], on the arc therefore allows for an efficient computation of the subset of primitives that must be checked for interactions. This approach greatly speeds up the algorithm on large grids with many primitives, as the computation time is now dependent on the density of primitives but otherwise linear in their number.

The strength of an interaction between any two features is dependent on the ratio between their current distance and collision distance. For two primitives, the distance is simply the euclidean distance of their waypoints, being the center points of their rectangular footprints while the collision distance is the smallest distance the primitives can be apart without overlapping along the vector connecting their center points. As the primitives considered in this work are always axis-aligned, the collision distance can be computed using

$$d_{\mathrm{col}} = \min \left\{ l_{P,1} \sqrt{1 + \frac{d_2^2}{d_1^2}},\ l_{P,2} \sqrt{1 + \frac{d_1^2}{d_2^2}} \right\}$$

with d_1 and d_2 being the center distances along the corresponding axes. For primitive-arc distances similar definitions are made, with the reference point on the arc being the closest point on the arcs midline from the center of the primitive.

3.3 Network Optimization

While the requirements for connectivity to the network are fulfilled from the initialization, the placement of the features is not yet optimized. The goal of the algorithm presented in this section is to generate a valid network, fulfilling the overlapping requirements specified in Section 2.2.2 and the minimum arc angle discussed in Section 2.4 while maintaining the initial structure. For this, an iterative, stage-based algorithm is employed. The optimization is performed in multiple stages, evolving from a fast, global approach to a more detailed and localized finalization. Each stage is applied at least once and can be repeated for multiple iterations if required. Iterations consist of two phases, a force-based discrete placement optimization followed by a network modification.

The first phase attempts to generate a valid network by iteratively moving the contained primitives. For this, a number of component forces are determined, each addressing a different violation. The resulting sum is then used to update the position of each primitive. After a number of repetitions, this approach converges towards a minimum of the combined force fields, terminating the phase. If this optimization does not result in a valid network, it is assumed that the network itself must be modified in order to achieve an acceptable result. For this, the second phase determines whether any arc should be split by a corner primitive, and whether any existing corner is superfluous and can be removed. Afterwards, this process is repeated in the next iteration until the network is valid or the computation aborted.

3.3.1 Placement Optimization

The presented algorithm determines up to six different force components to address the geometric requirements and optimizations presented throughout this work. Three components address overlaps of primitives and arcs as discussed in Section 2.2.2, while one enforces the minimal arc angle requirement stated in Section 2.4. The final two components further optimize the network by removing unnecessary corners and regulating arc lengths.

Collisions, resulting in overlaps of footprints, are addressed by the primitive-primitive, primitive-arc and arc length force components. All are determined similarly, using the collision grid and distance calculations established in Section 3.2. If the distance d between two components is smaller than the upper interaction limit $d_{\max}$, a separating force is computed following a continuous function

$$F(d) = F_{\max} \max\left\{ 0,\ \min\left\{ \frac{d_o - d}{d_o - d_i},\ 1 \right\} \right\}$$

with

$$d_i = d_{\text{col}} f_i \quad \text{and} \quad d_o = d_{\text{col}} f_o \quad \text{for} \quad f_i \leq 1,\ f_o \geq 1$$

This results in a linear ramp from 0 for the outer limit $d \geq d_o$ to $F_{\max}$ at the inner limit $d \leq d_i$. This continuity is necessary to smooth the resulting acceleration and prevent small asymmetries resulting from numerical errors to affect the overall system.

In order to still ensure that any violations resulting from an overlap are removed efficiently, it is important to choose f_i and f_o in a way to still have a significant separating force around the collision distance d_{col}. Choosing $f_o > 1$ however is unfeasible in this context, as any parallel, adjacent lanes would then be affected by a constant separating force. For this reason, the tolerances for all but the last stage of the presented algorithm are adjusted such that $f_i = 0.9$ and $f_o = 1.0$ can efficiently move any violations to be within tolerance. Only during the last finalization stage, a shorter ramp as shown in Figure 3.6 is employed to ensure that, in combination with the post-processing, the network validity is fulfilled according to the definitions in Section 2.2.2. To ensure stability and prevent oscillations from building up, a smaller maximal force is applied for this finalization. An additional benefit of this arrangement is that all violations above a certain overlap produce the same magnitude of force. As all violations must be removed eventually, they can be considered equally during placement phase and instead be further distinguished during the network modification.

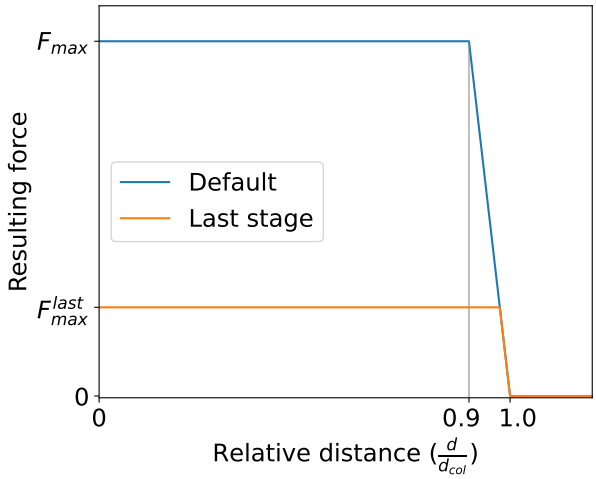


Figure 3.6 Profile of separating forces applied to overlapping components. The last stage employs a smaller ramp and force magnitude to achieve improved tolerances without becoming unstable.

For primitive-primitive collisions, the force is applied equally in opposite directions to both primitives. Forces opposing collisions of primitives and arcs are applied along directions normal to the arc and in full magnitude to the free primitive, while the tail and head of the arc receive a partial force, weighed by distance to the collision point on the arc similar to a mechanical rigid body. This approach is chosen as the movement required of an end primitive to resolve a violation increases with the length of the lever, potentially affecting otherwise unrelated areas of the network. If one of the primitives connected to the arc is fixed however, it is necessary to apply the full force to the free end to resolve the collision.

The force applied along an arc due to a violation of its minimal length follows the same structure, but is only applied if the minimal length l_{col} is longer than one index length, as it is otherwise redundant with the primitive-primitive collision force. If it is applied, both primitives are pushed apart along the direction of the arc, with $d_{\mathrm{col}} = l_{\mathrm{col}}$ being defined as the minimal required length. In this case, choosing $f_o > 1$ is viable, but not considered in the presented algorithm to maintain consistency.

Section 2.2.2 states an additional condition for a valid network, requiring that no two arcs that do not share a tail or head primitive may overlap. In the interest of efficiency, this requirements has not been assigned a separate force component in this phase, but is instead implicit in the primitive-arc force. If a network is planar, with no two arcs already overlapping, this is sufficient, as any arc forced to shift over another does so by first overlapping one of the connected primitives. Furthermore, the calculation of such a force component can not make use of the collision grid, but requires checking all arcs against each other, which is nonlinear in their number.

Two of the force components are related to the relative angles of any arcs connected to the same primitive. The validity requirements stated in Section 2.4 impose a minimal angle of $\alpha_{\mathrm{min}} = 90°$ to minimize overlaps, while as an optimization all corners should be minimized or removed. To enforce this, a separating torque is applied to each pair of arcs with insufficient relative angles α_Δ, following the continuous function

$$T(\alpha_\Delta) = T_{\mathrm{max}} \max\left\{0,\ \min\left\{\frac{(\alpha_o - \alpha_\Delta)}{(\alpha_o - \alpha_i)},\ 1\right\}\right\}$$

Similar to the collision forces, this results in a linear ramp from 0 for $\alpha_\Delta \geq \alpha_o$ to T_{max} at the inner limit $\alpha_\Delta \leq \alpha_i$. Following the same reasoning as above, while it is necessary to still have a significant torque around the angle required for validity, choosing $\alpha_o > \alpha_{\mathrm{min}}$ is not feasible as this would prevent the formation of rectangular structures such as grids or T-sections. The presented algorithm therefore defines $\alpha_o = 90°$ and $\alpha_i = 75°$, only increasing α_i for the final iteration to ensure validity while maintaining a continuous function.

If four arcs connect to the same primitive, the torque is applied only between immediately adjacent arcs, the opposing arcs do not directly affect each other to prevent redundant torques. As discussed for cross primitives in Section 2.4, a primitive with four connected arcs following this requirement forms a rigid structure with parallel opposing arcs. Enforcing this rigidity from the outset of the optimization however is infeasible, as relative angles are not considered by the network initialization algorithm presented in Section 3.2.

The second torque component is applied to straighten lanes along corner and interface primitives. While corners can be removed if the connected arcs are parallel, the performance of interfaces can also benefit from allowing a mover to travel on a straight trajectory to or from the station. As this component is not related to the validity of the network, it has a lower priority and magnitude than the components defined above. The resulting torque is shown in Figure 3.7 and chosen such that relative angles between $90°$ and $135°$ are not affected by either component. This improves the robustness of the optimization, as many solutions rely on angles within this range and would be negatively affected by torques towards either direction.

Torque components are converted to forces acting on the tail and head primitives of the affected arcs for the position update. For this, each arc applies a force with equal magnitude normal to its center line to both primitives, in the direction matching the torque. Two arcs with equal magnitude, but opposing torques, connected to the same primitive both apply this force, resulting in an addition of the forces towards the bisector of the arcs. As the other end primitives are additionally pushed apart, this results in an increase of the angle between the arcs.

The final optimizing force component again concerns the arcs' lengths relative to their index lengths. As waiting primitives are only placed at spacings of the index length, any additional distance of the arc cannot be used to increase the length of the queue but must still be traversed, increasing overall travel times and the size of the required footprints. For this reason, a force is applied to promote arc lengths towards integer multiples of their index lengths as shown in Figure 3.8. The step optimization force is chosen continuous and always acts towards the closest integer multiple of the index length, increased by a small margin. As during post-processing, a flooring operation of the relative length determines the number of possible wait positions, this margin allows for the compensation of small forces which would otherwise result in a length just below the next integer multiple. The chosen force function results in very small forces being applied to arcs between two index lengths multiples. While a bias towards one of the directions can be justified by either reducing the footprint or increasing the number of waiting spaces, the presented algorithm will not optimize these aspects unless some other component results in a bias towards one of the directions.

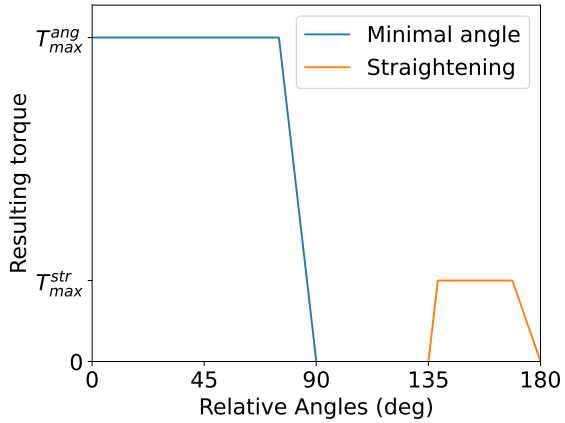


Figure 3.7 Torque components related to minimal angle violations and straightening optimizations for a pair of arcs connected to a corner, acting to separate the arcs.

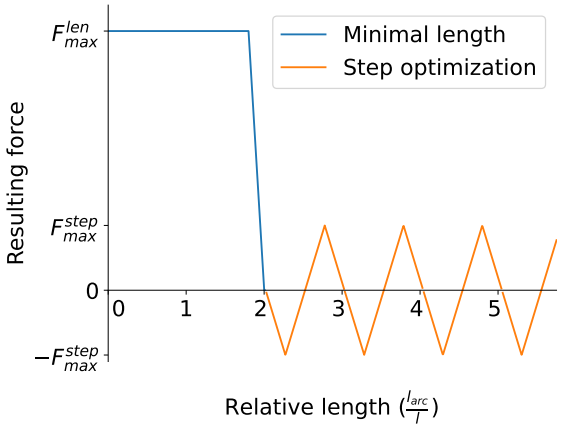


Figure 3.8 Force components related to collision violations and step optimizations for an arc with minimal length $l_{\text{arc}} = 2l$. Positive values act to lengthen the arc.

After all force components of an iteration are computed, the position of the non-fixed primitives is updated. The presented approach makes use of an intermediate velocity, resulting in a second order relationship between the sum of the force components and the primitive positions. This, in comparison to a first order relationship, is more robust towards frequently changing forces, which could otherwise result in an oscillation, while allowing a buildup of velocity for primitives affected by a consistent force. This relationship is expressed as

$$v_k(P) = v_{k-1}(P) f_{\text{friction}} + F_\Sigma(P) f_{\text{step},k}$$
$$p_k(P) = p_{k-1}(P) + v_k(P)$$

for the primitive P at the step k. The initial velocity $v_0 = 0$ is reset after every iteration and limited to $v_{\max}$, dependent on the current state. This limitation aims to prevent collisions with high velocities resulting in an overlap greater than the collision distance d_{col}, which would lead to an inversion of the force direction and allow the primitive to pass through the obstacle as discussed in Section 2.2.2 and visualized in Figure 2.3. For similar reasons, the friction factor f_{friction} decays the existing velocity, preventing an excessive buildup over long acceleration distances. The border of the field is modeled such that no primitive may pass it. In this, it behaves similar to the frame proposed by Fruchterman and Reingold [43], removing the remaining translation normal to the border after contact is made. Due to the axis-aligned, rectangular nature of the field, this is simply implemented by clamping the coordinates of all primitives to remain within the field after the position update.

The step size $f_{\text{step},k}$ is an important parameter of the algorithm, significantly influencing the robustness and speed of convergence. The presented algorithm varies the step size over the steps as shown in Figure 3.9, with different profiles for each of the stages discussed in Section 3.3.3. During the initial stages, it is possible that a network modification results in arrangements with significant overlaps, which generate large forces on the involved primitives until a sufficient clearance has been reached. To prevent an excessive buildup of velocity, the step size is limited during the first n_{init} steps until all such primitives have separated, before remaining constant until the end of the phase. While a reduction by a constant factor is a straightforward approach, this results in a step function and a corresponding jump in force magnitude at the step. For this reason, the presented algorithm makes use of a quadratic increase, which has shown to result in a reduction of the peak forces encountered. As the final stage of the optimization is only intended to perform minor changes to the positions and converge the primitives towards their final positions, a decaying step size is instead used to match the decreasing remaining error. An exponentially decreasing function has proven suitable for this, as it does not put a limit to the length of the final stage, allowing the resulting solution to be refined until the desired precision is achieved.

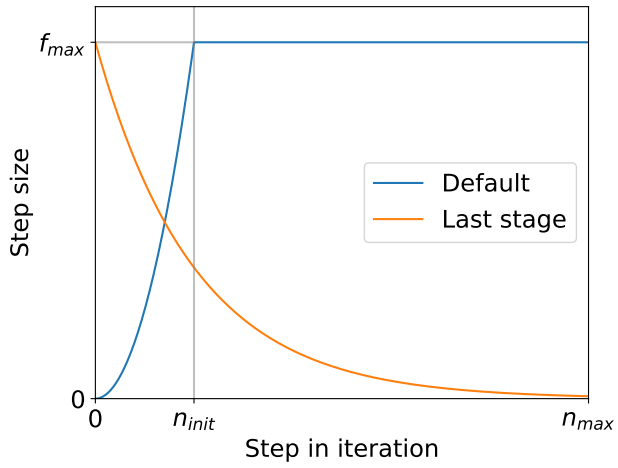


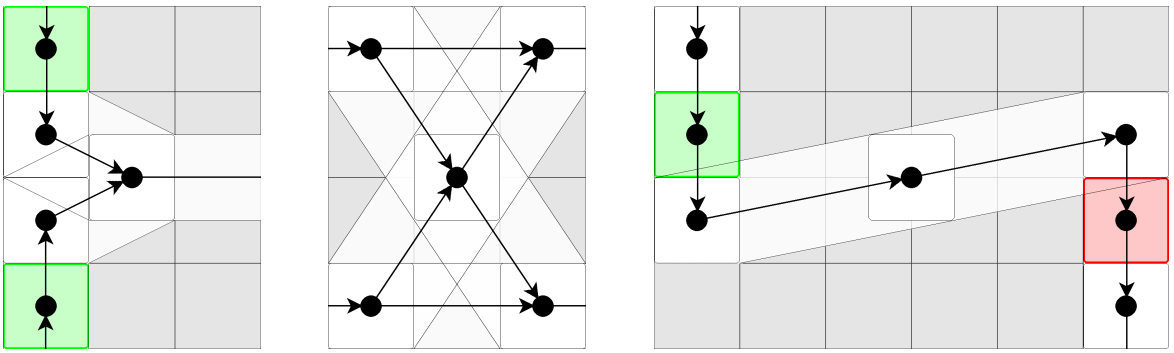
Figure 3.9 Step size used in the position update over the steps in each iteration. The default iterations limit the initial size to prevent excessive velocities, while the last stage converges towards the final positions.

3.3.2 Network Modification

After the placement optimization of the first phase has converged sufficiently close to a minimum of the forces, the network is tested for validity. The required level of convergence, dictated by the thresholds of the algorithm flow logic as discussed in Section 3.3.3, is dependent on the problem statement and must be empirically optimized to balance the efficiency and the robustness of the optimization.

If the network is not valid, it is assumed that the current network has no reachable arrangement in which it is valid and must be modified to continue the optimization. The presented algorithm makes use of two operations, the addition and the removal of corners. These modifications are straightforward to employ, as they only affect the local area, either a single arc or a corner primitive with two connected arcs, and do not alter the global structure of the network such as the connectivity of paths and sections. The modifications are performed such that no additional violations, such as new arc intersections, are introduced. The first of the stages introduced in Section 3.3.3 has a special function during the modification phase, as only during this stage additional corners can be introduced. After completion of the first stage, the network is assumed to have all corners required to be solvable.

As the first operation of each modification phase, all qualifying corners are removed by modifying the network as described in Section 3.2. Corners qualify if either connected arc is shorter than its corresponding index length or the angle between the connected arcs is above a given threshold. Additionally, during the first stage, qualifying corners cannot be subject to significant torque components. During this stage, the torques resulting from an angle violation introduced in Section 3.3.1 frequently indicate that the amount of corners within the neighborhood of the arc are not sufficient for right angles. Examples of this are shown in Figures 3.10a and 3.10b, in which the existing primitives and arcs cannot form right inner angles. In other, more complex arrangements such as in Figure 3.10c, a single corner is not sufficient to create a valid arc, requiring two modification phases to reach a solvable arrangement. In this case, removing the corner is counterproductive and could result in a loop of continuous additions and removals.



(a) Triangle resulting from adjacent interfaces.

(b) Triangles resulting from a bypassed cross.

(c) Arc requiring two corners to be viable.

Figure 3.10 Network arrangements leading to configurations which cannot be solved without additional corners.

As the second step of the modification, after all qualifying corners have been removed, the remaining arcs are considered for splitting. To reduce the number of unnecessary modifications, arcs that have just been merged and arcs shorter than or near their index length are not considered. Additionally, arcs that connect two fixed primitives and are adjacent to the field border are considered fixed and excluded from splitting. For all remaining arcs, a number of metrics are determined to evaluate the usefulness of a split. The metrics are tested in sequence, if any arc of the current metric exceeds a given threshold, this metric is employed and the remaining discarded. Then, all arcs within a small tolerance of the largest reached score are split using the method introduced in Section 3.2. This process, in contrast to splitting a fixed number of arcs, allows the algorithm to retain any symmetries in the network despite numeric instabilities. If no arc for any metric exceeds the threshold, no arcs are split and the phase is completed.

During the optimization, primitives are generally pushed apart until they collide with either the field border or other arcs, which in turn are connected to constrained primitives. During a collision, it cannot be guaranteed that the primitives connected by the arc are close to the colliding primitive, in either geodesic or euclidean distance. For this reason, designing a transitive metric determining the arc whose split is ultimately required to generate sufficient space for a valid network is infeasible. The presented algorithm instead makes use of an iterative approach, attempting to resolve the required splits "inside-out", while prioritizing overlaps over arc angle violations, over performance optimizations. As arc angle violations are generally more localized, while the resolution of overlaps typically has larger effects on the overall structure, this prioritization follows the overall strategy of optimizing the global layout before finalizing local structures. In this, changes implemented to resolve overlap resolutions have the potential to also resolve arc angle violations without the need for additional corners. Additionally, if substructures of the network, such as bypassed intersections, are shifted over large distances, smaller arrangements with less corners tend to be more rigid and able to retain their relative shapes, increasing the robustness of the overall optimization.

The presented algorithm makes use of two metrics which can be computed efficiently and simultaneously with the force components during the placement optimization phase. The primary metric is computed by the sum of all violations of overlap components, primitive-primitive, primitive-arc and arc length, discussed in Section 3.3.1. For this, the scores

$$S_{\mathrm{PP}} = S_{\mathrm{PA}} = (1 - \frac{d}{d_{\mathrm{col}}}) \geq 0 \quad \text{and} \quad S_{\mathrm{AL}} = (1 - \frac{l_{\mathrm{arc}}}{l_{\mathrm{col}}}) \geq 0$$

are summed to the overlap score

$$S_{\mathrm{Overlap}}(A) = S_{\mathrm{PA}}(A) + S_{\mathrm{AL}}(A) + S_{\mathrm{PP}}(Head(A)) + S_{\mathrm{PP}}(Tail(A))$$

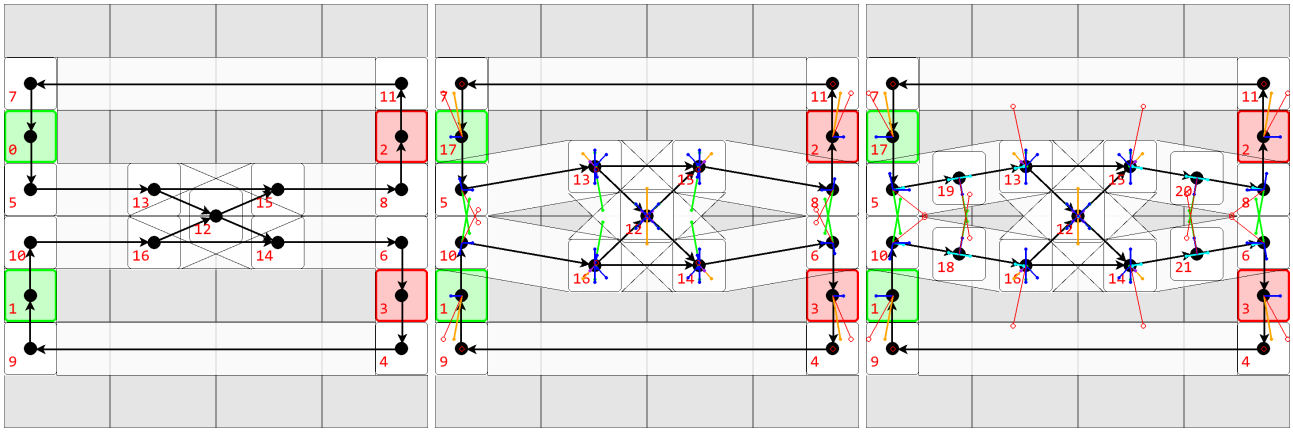
for all arcs A. If no arc exceeds the set threshold for this score, the secondary score also considers the arc angle violations using

$$S_{\mathrm{AA}} = (1 - \frac{\alpha_{\Delta}}{\alpha_o}) \geq 0$$

resulting in the violation score

$$S_{\mathrm{Violation}}(A) = S_{\mathrm{Overlap}}(A) + S_{\mathrm{AA}}(A)$$

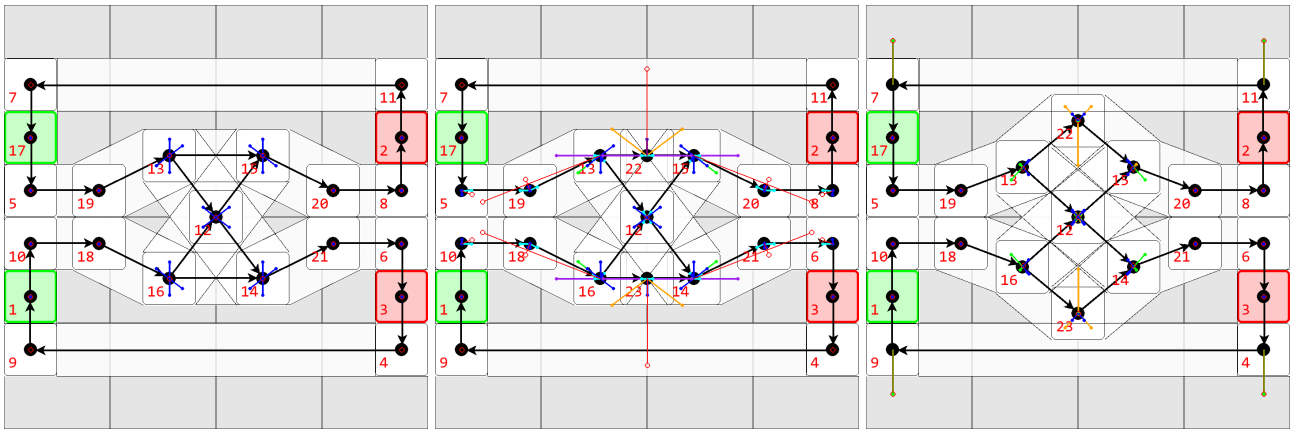
Figure 3.11 visualizes the effect of the metrics and the used "inside-out" principle resulting in an iterative expansion. Starting from the initial violations remaining from the first placement optimization shown in Figure 3.11b, the primary metric splits the arcs #5 $\rightarrow$ #13, #10 $\rightarrow$ #16, #15 $\rightarrow$ #8 and #14 $\rightarrow$ #6, prioritizing them over the arc angle violations of the primitives #13 through #16. After the second placement optimization shown in Figure 3.11d, all overlaps have been resolved. Therefore, the arc violations are approached by splitting the arcs #13 $\rightarrow$ #15 and #16 $\rightarrow$ #14. These new corners however result in new overlap violations shown in Figure 3.11f, which prompt the splitting and outward expansion of arcs #11 $\rightarrow$ #7 and #4 $\rightarrow$ #9. After the fourth placement optimization in Figure 3.11h, all violations are within the tolerances of the first stage, prompting the algorithm to proceed with more localized manipulations.



(a) Initialized network.

(b) First placement optimization. Arcs such as #10 → #16 overlap the stations and the arc angles along the inner interfaces, splits and merges are invalid.

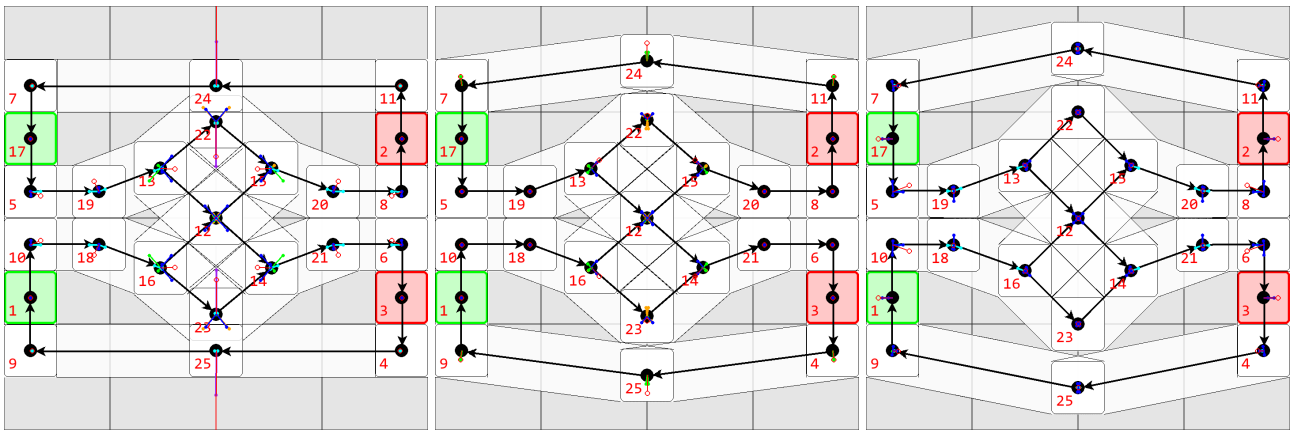
(c) First network modification. The overlaps are prioritized and split.



(d) Second placement optimization. The overlaps are removed, the invalid arc angles remain.

(e) Second network modification. Arcs #13 → #15 and #16 → #14 are split.

(f) Third placement optimization. The arc angles are now within tolerance, but #22 and #23 overlap the outer arcs.



(g) Third network modification. The outer arcs #11 → #7 and #4 → #9 are split to make space.

(h) Fourth placement optimization. The network is now within tolerance of the first stage, proceeding the optimization.

(i) Final network, after stages two through four.

Figure 3.11 Example of placement optimization and network modification phases of repeated first stage iterations. Forces are indicated by colored arrows. Overlaps are purple, green and orange, forces resulting from torques blue and the resultant amplified in red.

3.3.3 Iteration Stages

The phases presented in Sections 3.3.1 and 3.3.2 are applied to the network in an alternating fashion, which is repeated until a valid network is reached or the optimization aborts after failing to find a solution. As shown in Algorithm 2, the optimization is structured in iterations, with each consisting of a placement optimization followed by a network modification phase. While each iteration is structured similarly, the actions necessary at the beginning of the optimization differ significantly from the actions finalizing the network. This change is represented through four stages, each defining the behavior of the contained iterations. During a full run, the network progresses through each stage successively, with later stages increasing the number of enforced requirements or reducing the accepted violation tolerances. This iterative introduction improves the performance of the algorithm, as it allows for aspects more relevant to the global network structure to be optimized first without being affected by aspects which can only be reasonably optimized after the local neighborhood is sufficiently established. Applying all aspects from the outset has the potential to disrupt the optimization by forcing the network towards a local optimum which does not meet the requirements for global validity. An additional benefit of this approach is that the step size can be increased in the earlier stages, while the final stage refines the solution using small steps and tolerances as discussed in Section 3.3.1, thus reducing the total number of steps necessary.

As means to control the placement optimization phase, each stages defines, following the notation of Algorithm 2, a maximal number of steps $step_{\max}$ and a more refined STEPBREAKCONDITION function, allowing an early abort of the first phase. This condition is intended to predict the step at which the network is closest to the currently reachable force minimum, or after which the optimization will not make any more significant process towards the minimum. Additionally, each stage defines different magnitudes and tolerances for each force component or disables them entirely. For the step update, each stage defines a function for the step size f_{step} as well as friction factor f_{friction} as discussed in Section 3.3.1.

Algorithm 2 Network optimization algorithm, with the behavior of each iteration dependent on the current stage.

$stage \leftarrow$ FIRSTSTAGE
for $iter \leftarrow 0$ **to** $iter_{\max}$ **do**
  /* Placement optimization phase */
  for $step \leftarrow 0$ **to** $step_{\max}(stage)$ **do**
    GETFORCES($stage$)
    UPDATEPOSITIONS($stage, step$)
    if STEPBREAKCONDITION($stage$) **or** FORCELIMIT **then break** $step$
  end for

  /* Network modification phase */
  REMOVECORNERS()
  if $stage$ **is** FIRSTSTAGE **then**
    if $\max(S_{\text{Overlap}}) > threshold$ **then**
      SPLITARCS(S_{Overlap})
    else if $\max(S_{\text{Violation}}) > threshold$ **then**
      SPLITARCS($S_{\text{Violation}}$)
    end if
  end if

  if STAGECOMPLETECONDITION($stage$) **then**
    $stage \leftarrow next(stage)$
  end if
end for

As discussed in Section 3.3.2, only the first stage is capable of introducing new corners, after which the network is assumed to have all required elements and no additional primitives are added by the later stages. For this, a primary and secondary metric with corresponding thresholds for activation are defined. While all stages are capable of removing corners, this distinction requires the first stage to define a separate threshold angle above which corners may be removed, as well as a limiting torque, above which corners are exempt from removal.

If at the end of an iteration, the network does not meet the provided STAGECOMPLETECONDITION function, the current stage is repeated. Otherwise, the stage is advanced, while after the final stage the optimization is completed, as the post-processing is capable of producing the validity of the network. The algorithm flow logic makes use of a number of metrics, which quantify the current state of the various optimization steps. As the final goal is the minimization of all violations, a straightforward approach is to consider the largest current violation score as defined in Section 3.3.2. From this follows the condition

$$\text{VIOLATIONCUTOFF} \leftarrow \max \{ S_{\text{Violation}}(A_k) \; \forall_{A_k} \in A \} \leq \text{violation}_{\min}$$

which is computed alongside the component forces and during the network modification using a stage-dependent threshold $\text{violation}_{\min}$.

As the position update of placement optimization makes use of a velocity in a second-order relationship as discussed in Section 3.3.1, the condition

$$\text{INITCOMPLETE} \leftarrow step \geq step_{\min}$$

ensures that a number of steps have passed after the initialization with $v_0 = 0$ and the velocity has sufficiently stabilized for the related metrics to be considered accurate.

The next set of metrics predict the point in the optimization after which no significant additional progress will be made towards the force minimum. This improves the efficiency of the algorithm, as computing any further steps would have no added benefit or even degrade the solution. For this, the largest velocity of any primitive is monitored following

$$\text{VELOCITYCUTOFF} \leftarrow \max \{ v(P_k) \; \forall_{P_k} \in P \} \leq v_{\min} \wedge \text{INITCOMPLETE}$$

stopping the placement optimization if no primitive moves faster than the threshold $v_{\min}$ after the initial acceleration. This maximum metric has superior performance over an average or mean velocity, as the network modification frequently resolves violations only affecting their local neighborhood. In this case, the majority of the network remains stationary, while a small number of primitives is repositioned over large distances. The metric however performs poorly when primitives are oscillating or otherwise move quickly within a limited area. For this case, the movement difference metric is employed, which approximates the largest distance between any two positions of a primitive within the last m steps. While this introduces a delay into the detection, it relies on the overall trend of the movements and is therefore more robust to non-productive high velocities. As computing an exact solution to this for all primitives in every step is not feasible, it is approximated using the euclidean distance to the position of the primitive at the start of the iteration following

$$\begin{aligned} moveDiff(P,n) = &\max \{ \|p_i(P) - p_0(P)\| \; \forall_i \in \{ n-m, \ldots, n-1 \} \} \\ &- \min \{ \|p_j(P) - p_0(P)\| \; \forall_j \in \{ n-m, \ldots, n-1 \} \} \\ \text{MOVEDIFFCUTOFF} \leftarrow &\max \{ moveDiff(P_k, n) \; \forall P_k \in P \} \leq p_{\min} \wedge \text{INITCOMPLETE} \end{aligned}$$

for the current step n and the threshold $p_{\min}$, which can be implemented efficiently by storing the computed distances of the last steps in a queue or similar FIFO data structure. While this does not capture movement normal towards the total traveled path, such cases have not been found to occur frequently in practice. An

alternative, but more expensive, approach addresses this by relating the distances to the average position over the last m steps.

A limiting factor of the step size, and therefore the speed of convergence, is the tendency of primitives to oscillate after reaching stable positions, similar to a driven mechanical oscillation. While a number of measures are in place throughout the presented algorithm to prevent this, the FORCELIMIT condition is used for safeguarding against already converged networks diverging due to these effects. For this, two different metrics can be employed. As a straightforward measure, the largest allowed force can be limited by a constant threshold, ending the phase if the condition

$$\text{FORCELIMIT} \leftarrow \max\{ F_\Sigma(P_k)\ \forall_{P_k} \in P \} \geq F_{\max} \wedge \text{INITCOMPLETE}$$

is fulfilled. This ensures that, after all initial violations have been sufficiently resolved during the initialization, the optimization can be halted as soon as a force above the threshold $F_{\max}$ occurs due to excessive oscillation. This metric however requires careful tuning of the threshold to detect diverging networks early without halting the optimization early during normal operation. An alternative metric solves this by introducing a dynamic limit, based on the lowest maximum force of the iteration. Using

$$\begin{aligned} F_{\max,\text{step}}(n) &= \max\{ F_\Sigma(P_k)(n)\ \forall_{P_k} \in P \} \\ F_{\max,\text{dyn}}(n) &= f_{\text{dyn}} \min\{ F_{\max,\text{step}}(n)\ \forall_n \in P\{0,\dots,n\} \} \end{aligned}$$

the dynamic threshold $F_{\max,\text{dyn}}$ for the step n can then be used instead of the constant $F_{\max}$ to detect if the force has increased by a factor of f_{dyn} over the lowest value encountered during the iteration, indicating a significant increase in a violation and therefore a divergence from the solution.

The presented algorithm makes use of four different stages, each prioritizing a different aspect of the optimization. The first placement stage optimizes the global structure of the network, which typically has large violations and unbalanced positions introduced in the initialization. Additionally, this is the only stage capable of introducing new corners, such that after completion of this stage the network is assumed to have all required primitives to converge to a viable arrangement. The stage is only required to roughly establish the positions, neighborhoods and substructures of the final network. For this, large step sizes and tolerances are used to efficiently move primitives over large distances, while optimization components such as the corner straightening and arc length steps are not considered.

As the modification phase of each iteration of this stage can only address a single violation, the stage must potentially be repeated multiple times, with each repetition significantly altering the overall structure. For this reason the STEPBREAKCONDITION of this stage is defined as

$$\text{STEPBREAKCONDITION} \leftarrow \text{VELOCITYCUTOFF} \vee \text{MOVEDIFFCUTOFF} \vee \text{VIOLATIONCUTOFF}$$

with large tolerances on the VELOCITYCUTOFF and MOVEDIFFCUTOFF to quickly converge to an approximate solution sufficient for removing another violation. The VIOLATIONCUTOFF, which in this stage additionally serves as the STAGECOMPLETECONDITION, ensures that the next stage is entered as soon as the violation is deemed solvable by halting the ongoing placement optimization early.

The second and third stage of the algorithm are structured similarly and further refine the arrangement of the primitives. For this, both stages make use of smaller step sizes than the first while enforcing tighter tolerances on the overlaps and minimal angles. Specifically, the second rotation stage is intended to refine all arc angles, removing remaining small angle violations and making use of the straightening torque component to remove unnecessary corners. Afterwards, the third shifting stage further optimizes the relative positions of the primitives by including the arc step length component. The introduction of these components typically results in closely packed primitive arrangements, forming localized grids with parallel arcs where appropriate. The flow logic functions are both defined equally as

$$\text{STEPBREAKCONDITION}, \text{STAGECOMPLETECONDITION} \leftarrow \text{VELOCITYCUTOFF} \vee \text{MOVEDIFFCUTOFF}$$

In this, the network is continuously converging while the maximal number of steps $step_{\max}$ regularly halts the placement optimization and allows the network modification phase to remove all qualified corners. Only after the network has sufficiently converged, the next stage is entered.

The fourth stage is intended to finalize the optimization by converging the primitives towards their final positions. For this, a decaying step size and smaller force ramp is employed as discussed in Section 3.3.1. Additionally, all tolerances are chosen as small as viable without generating excessive instabilities. The goal of this stage is to reduce the remaining error to within a level which is resolvable by the operations of the post-processing defined in Section 3.4. For this, the STEPBREAKCONDITION solely consists of the VIOLATIONCUTOFF, while no STAGECOMPLETECONDITION is defined. Instead, the optimization step of the pipeline is completed only after either the VIOLATIONCUTOFF is valid and the network modification phase has no remaining qualifying corners, or the maximal number of iterations $iter_{\max}$ is reached.

3.4 Post-Processing

As the final step of the offline pipeline, the optimized network is cleaned up and prepared for use by the controller. For this, numeric inaccuracies remaining from the optimization are removed and waiting primitives, sections and sequences generated. Finally, the network is converted to a graph.

As discussed in Section 3.3.1, the placement optimization requires a significant force along an outline of the primitive footprints to ensure that no collision is present in the final placement. This however is not viable for typical environments, as it would result in separating forces being applied to adjacent primitives and arcs, preventing the formation of closely packed arrangements. For this reason, the optimization can only converge the positions of primitives to within an error to a valid state. The first step of the post-processing addresses this remainder by rounding the position coordinates and fitting all primitives to the closest point on a coordinate grid. The smallest viable size of this grid is dependent on the performance of the finalization stage introduced in Section 3.3.3, but typical values are small fractions of the primitive edge lengths, ranging from $5\,\mathrm{mm}$ to $20\,\mathrm{mm}$. An additional benefit of this rounding is that floating-point values and their associated artifacts are removed, simplifying the detection of parallel arcs. While qualified arcs within a small tolerance have already been merged during the optimization, this discrete placement allows for exact parallelism.

After discretizing the primitive positions and arc lengths, arcs longer than two index lengths l can be split by introducing waiting primitives. For this, each arc of length l_{arc} is split by $\lfloor \frac{l_{\mathrm{arc}}}{l} \rfloor - 1$ waiting primitives placed along the centerline at equidistant points l apart, starting from the head of the arc. This placement ensures that all waiting positions are as far forward along the lane as possible. At this point, the primitives and arcs of the network are final. This allows the sections defined in Section 2.2.2 to be generated, based on the continuously updated paths introduced to the network in Section 3.2. As each path containing a section also contains all arcs of the section, the sections of a path can be computed by splitting the path at all merge and split type primitives. Each section created this way inherits the set of managed item types from the path, if multiple paths generate the same section the union of item types is formed. As a final step, the item sequences of the network can be generated following the process discussed in Section 2.2.4.

Much of the data generated during the offline solution generation is not required by the online control. In the interest of encapsulation and promoting compatibility with other solution generation processes, the final output is given in the form of a graph as defined in Section 2.2.3. Following the discussed procedures, the components of the network are converted to their graph equivalents, only retaining the information required by the controller. For this, the primitives are reduced to nodes, representing waypoint coordinates for mover navigation with all geometric information contained in the awaitable and skippable properties to be used as discussed in Chapter 4. While all primitives may be marked as awaitable, the performance of the network can be improved by excluding critical sections such as intersections. For this reason, all cross primitives and all primitives along the connecting section of a merge-split are assigned the non-awaitable status by the presented algorithm. If the network includes arcs shorter than their index length, additional care must be placed when assigning the awaitable property to prevent intersections of the waiting positions. All primitives with parallel incoming and outgoing arcs are marked as skippable, as they can be traversed by a mover in a straight line without decelerating. If the primitive connects multiple lanes, the property can be defined individually for each valid combination of incoming and outgoing arc.

4 Concepts of the Online Control

Programmable logic controllers, or PLCs, are commonplace in modern industrial assemblies. While most modern PLC-like controllers use general-purpose PC hardware, their capabilities are limited compared to modern workstation PCs using high-level languages. As they are designed for use in environments with hard real-time requirements, they typically rely on a cyclic execution model with fixed cycle times allowing for an easily predictable worst-case reaction time. In this however, they operate less resource efficient than machines without this requirement, operating on e.g. an event-driven execution model. Additionally, the development of applications for PLCs requires extensive dedicated hardware or simulators. For these reasons, the controller implementation referenced in this work makes use of the high-level language Python [49] and is employed on a workstation PC. Despite this, the controller is designed such that it could viably be implemented on a PLC adhering to the IEC 61131 standard. This choice significantly limits viable algorithms and is a key factor for the extensive offline preprocessing discussed in Chapter 3, as well as the design of the item sequences presented in Section 2.2.4. A simulation of the environment is used to verify both the controller and the generated solutions using the software-in-the-loop methodology. This approach allows for an autonomous testing with faster than real-time execution, while scenarios can be specified to provide exact repeatability. The simulator allows the generated sequences to be visually presented as shown in Figure 4.1, allowing for easier supervision and demonstrations of the planned system.

The considered planar motor systems make use of a dedicated controller, such as the Planar Motor PMC [50] for controlling the field actuators and sensors. In this, all dynamic considerations and the supervision of the movers are abstracted into an interface, typically connected via a field bus to the overall station controller. These interfaces accept two types of control sequences to be performed, either a chain of waypoints for linear point-to-point movement or detailed six degree of freedom trajectories. The presented controller is therefore intended to operate in cooperation with PMC by providing node locations as waypoints for linear point-to-point movement, only directly dictating mover trajectories if more complex movements are required for station interactions. This allows the controller to use all resources for path planning and managing the interactions with other components of the overall assembly, including the input and output feed stations. While the PMC integrates a basic level of collision prevention, the presented controller implements both more preventative collision and deadlock avoidance measures. Additionally, the controller must ensure that other requirements specified in Chapter 2, such as the station interaction sequences, the robustness to variances in interaction duration and the maximal changeover times of movers at stations while maintaining a minimal throughput are met.

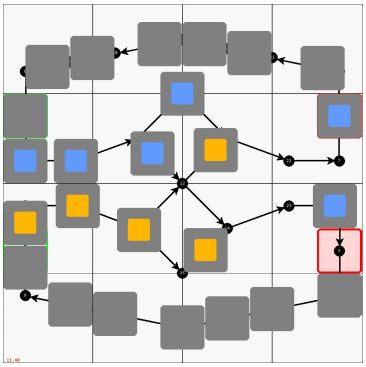


Figure 4.1 Example of a visualized sequence performed by the simulated real-time controller. The graph is superimposed by grey movers, carrying alternating blue and orange items to the receivers.

4.1 Simulator Environment

The presented simulator is used for software-in-the-loop testing of the controller and generated solutions, providing interfaces for both the interaction with the lower-level PMC controller and to adjacent systems within the overall assembly. Each run of the simulator is based on a scenario, which contains the initial environment specifications, available number of movers and runtime of the simulation. Additionally, the degree of randomness of the station interactions can be specified and seeded for repeatability. The actions of the controller and stations during the test are monitored and compiled into a sequence, which can then be visualized in an interactive GUI or as a video file.

To simulate the interaction with a station, the mover is released by the controller to the simulation and returned after the specified interaction time τ_I, carrying the modified item type. During the interaction, the controller is required to maintain a reference to the mover and prevent all other movers from entering the station. As the stations of the physical system are not required to track the passed items or to provide an interface communicating the loaded item type, the controller must additionally keep track of the expected interactions and loading states of each mover during runtime. While the simulation supervises the controller by comparing the tracked state to an internal reference, the controller is not notified if a wrong item is delivered. Additionally, the simulation logs the item delivery rate to each station for later analysis. For this, a startup phase is defined until the first receiver interaction occurs, after which the number of delivered items and time until the end of the simulation are recorded. This approach allows for a reliable estimate of the long term expected throughput of the overall system and each individual station, even for relatively short simulation durations. These collected metrics can then be used to compare the performance of different solutions for an environment.

In order to increase the efficiency of the simulation, a number of aspects of the real system are abstracted. The simulation does not model the dynamics of each mover during runtime, but instead approximates the traversal durations for each action according to the dynamics discussed in Section 2.3. As a simplification, the simulation does not consider the change in traversal duration due to acceleration and deceleration. As this negates the differences between continuous and point-to-point movement, this affects the performance metrics used to compare solutions and must be considered when comparing the simulation to an execution on the physical system. Furthermore, abstracting the mover dynamics prevents the simulation from performing additional collision checking as would be the case when using the PMC. Instead, the design of the path planning is laid out conservatively, ensuring that no collisions are generated for solutions with a valid network and correct assignments to the nodes' awaitable and skippable properties according to the definitions in Section 4.2. In the physical system, a less conservative and more efficient approach is viable by underestimating the traversal times and required clearances, while relying on the PMC to perform the fastest possible execution of the planned trajectories while avoiding collisions.

As both the simulation and controller are implemented on an external computer, it is possible to perform the simulations faster than real time. For this, the simulation makes use of the mover state tracking discussed in Section 4.3, which either places a mover into a queue ordered by the deadline of the next planning operation, or creates a dependency on another blocking mover. In the latter case, the blocked mover is scheduled for planning after the blocking mover has cleared the blocking condition. While a real-time system would always start the next computation in the queue after completion of the previous item or idle if the queue is empty, the simulation immediately skips the simulation time to the deadline of the next required operation to perform all computations and dependency resolutions without any simulated time passing. This approach however requires the simulation to supervise the computation duration to ensure that all deadlines would be kept during a real-time operation.

4.2 Action Planning

The central element of the controller is the action planning algorithm presented in this section. After determining the chain of edges to be traversed by the mover using the control strategy described in Section 4.3, the controller employs the Algorithm 3 to generate a tuple of actions containing the planned trajectory. Each action represents a point-to-point movement traversing a single edge, defining a start time t_{start} at which the mover starts leaving the tail node of the edge and, based on the duration of the edge, an end time t_{end} at which the mover becomes stationary at the head node. Two actions along incoming and outgoing arcs of a node can therefore immediately follow each other. While the physical system could benefit from combining actions along skippable nodes to prevent unnecessary stops and reduce the traversal duration, the simplified simulation does not consider accelerations and decelerations of movers as discussed in Section 4.1.

In order to avoid collisions, the controller only assigns actions to movers if they are considered safe. This is the case if it can be ensured that no other mover is occupying the planned edges or nodes during traversal and that the target of the final action is awaitable. If a node is awaitable, it indicates that it is safe for a mover to become stationary and remain there for some time without blocking other sections of the graph. While a mover is able to come to a stop on nodes that are not awaitable, it must have an immediately following outgoing action. Requiring that movers always plan until reaching a free, awaitable action target node ensures that, even if no other actions are planned for the mover before its arrival, the mover comes to a stop at a safe node and could remain there without affecting other parts of the graph. A node reserved as the action target of a mover is not considered safe to be entered by another mover until the reservation is released. This ensures that at no time the timely planning of an action is necessary to avoid a collision, as each mover has a safe reserved node available at all times. Initially, the action target node of a mover is the assigned initial position, while after each successful action planning operation this reservation is updated as shown in Algorithm 3.

To maximize the potential flow of the system, it is necessary to minimize the time any node or edge is reserved for a mover. This can be accomplished by each node tracking all actions passing through it, only permitting a new action if its required time slice does not overlap that of any already scheduled action, while compensating for overlaps. This has been shown to perform well [33], but requires a tracking of intervals for all planned actions. The presented controller employs a simpler approach by not permitting an interweaving of actions, which would allow a scheduling of an action ending before an already reserved slot. This limitation allows the node to only reference the end time of the last planned action in t_{nextFree}, instead of individually tracking all planned actions. Additionally, this design simplifies the sequence tracking discussed in Section 4.3, as the employed item sequences are only capable of tracking the next required set of items and do not support retroactive changes to the mover order. Each node has a precomputed clearing duration τ_{clear} according to the dynamics discussed in Section 2.3, representing the duration an incoming action must be delayed until an outgoing mover has cleared sufficient space. An action A entering a node $H = \text{Head}(A)$ is only considered safe if

$$t_{H,\text{nextFree}} + \tau_{H,\text{clear}} \leq t_{A,\text{end}} \tag{4.1}$$

holds, ensuring that the node has sufficient time to be cleared after the start of an outgoing action before the arrival of an incoming action.

If during planning, a mover is not able to safely reach the intended target node, it falls back to the last awaitable node if one exists and only schedules the actions leading to it. The next planning operation, attempting to generate actions for the remaining edges, is then scheduled based on the $blockingCondition$ as discussed in Section 4.3.

Algorithm 3 Action planning algorithm for the mover M along a tuple of edges. The algorithm assumes that the target node $\mathrm{Head}(E_n)$ is awaitable or that an immediate continuation of the mover is planned afterwards. The commitment to the planned actions and node reservation is shown for completeness, but not performed at this place when the algorithm is called by the presented controller, following the discussion in Section 4.3.

Input: Mover M
Start time t_{start}
Edges $E = (E_1, \ldots, E_n)$
Output: Actions $A = (A_1, \ldots, A_m) : m \leq n$

for $E_k \in E$ **do**
  $H \leftarrow \mathrm{Head}(E_k)$
  $t_{\mathrm{end}} \leftarrow t_{\mathrm{start}} + \mathrm{Duration}(E_k)$
  $A_k \leftarrow$ action traversing E_k from t_{start} to t_{end}
  if H is already action target of another mover M_B **then**
    /* M is dependent on M_B */
    $\mathrm{blockingCondition} \leftarrow$ MOVER M_B
    break E_k
  else if $t_{H,\mathrm{nextFree}} + \tau_{H,\mathrm{clear}} \leq t_{\mathrm{end}}$ **then**
    /* M arrives too early at H */
    $\tau_{\mathrm{wait}} \leftarrow t_{H,\mathrm{nextFree}} + \tau_{H,\mathrm{clear}} - t_{\mathrm{end}}$
    $\mathrm{blockingCondition} \leftarrow$ TIME τ_{wait}
    break E_k
  end if
  $A \overset{\mathrm{put}}{\leftarrow} A_k$
  $t_{\mathrm{start}} \leftarrow t_{\mathrm{end}}$
  if H **is** awaitable **then** $\mathrm{safeNode} \leftarrow H$
end for
if target is not reached **then**
  remove actions after $\mathrm{safeNode}$ from A
end if

/* Commit to actions by reserving nodes. This is instead
 * performed in PLANMOVER when called by Algorithm 5. */
for $A_k \in A$ **do**
  $t_{\mathrm{Tail}(A_k),\mathrm{nextFree}} \leftarrow t_{A_k,\mathrm{start}}$ for mover M
end for
move action target reservation for mover M from initial $\mathrm{Tail}(A_0)$ to final $\mathrm{Head}(A_m)$

4.3 Control Strategy

As discussed in Section 4.1, the controller is not implemented on a real-time capable hardware controller but within a simulation, executed on workstation PC. For this reason, the main loop laid out in Algorithm 4 demonstrates the simulated operation of the controller, making use of the discussed assumptions and modifications. The algorithm consists of a main loop executed by the simulation, which, dependent on the state of the next mover to be scheduled, either performs an interaction or calls the controller to plan the next actions of the mover. All movers either performing an action or waiting for a fixed time are eligible for scheduling. The presented simulation is capable of halting time during scheduling and is therefore always able to process each mover at the latest possible time for scheduling. As this leads to bursts in computational requirements, this is not feasible for a real-time controller with limited computational capacity, which is required to distribute the computational load over time. This can be accomplished by continuously computing actions according to a priority queue algorithm such as earliest deadline first, as the maximal computation duration and times at which each mover requires new actions are deterministic, or within a limit for stations with varying interaction times, and known to the controller.

Algorithm 4 Main loop of the simulator. The simulated controller is encapsulated, only having access to information that would be available during an employment in a physical system.

Input: Movers $M = (M_1, \ldots, E_n)$
Test duration $t_{\max}$
Output: Logs with action and interaction sequences
$t_{\text{now}} \leftarrow 0$
while $t_{\text{now}} < t_{\max}$ **do**
  $M_{\text{queue}} = \{\, M_k \in M \mid \text{State}(M_k)$ **is not** BLOCKED $\,\}$
  $M_{\text{next}} \leftarrow \min \{\, t_{M_k,\text{nextPlan}} \; \forall_{M_k} \in M \,\}$
  $t_{\text{now}} \leftarrow t_{M_{\text{next}},\text{nextPlan}}$
  if $\text{State}(M_k)$ **is** TOSTATION **then**
    /* Mover just arrived at station for interaction */
    $\text{State}(M_k) \leftarrow$ INTERACTING
    $t_{M_k,\text{nextPlan}} \leftarrow t_{\text{now}} + \tau_I$
    Log station interaction
  else if $\text{State}(M_k)$ **is** TOBLOCKED **then**
    /* Mover just arrived at goal of last planned action, is now blocked */
    $\text{State}(M_k) \leftarrow$ BLOCKED
  else if $\text{State}(M_k)$ **is** TOWAIT **then**
    /* Mover just arrived at goal of last planned action, is now waiting */
    $\text{State}(M_k) \leftarrow$ WAIT
  else
    /* Mover just arrived at goal of last planned action, plan new actions */
    PLANMOVER($M_{\text{next}}, t_{\text{now}}$) ▷ Defined in Algorithm 5
    Log new actions
  end if
end while

During the initialization of the system, each mover is assigned an initial position from which it is handed over to the controller. As all movers are empty at startup, their first interaction will be with a provider station. For this reason, the algorithm assigns all providers as initial positions, before working backwards along the incoming paths, assigning all awaitable positions to movers until the available number is reached. If there are not sufficiently many awaitable spaces along the paths and a receiver is reached before all movers are placed, the remaining initial positions continue following the paths ending in the receivers. If there are more movers than awaitable nodes, the initial positions can not be assigned and movers have to be removed or parking positions outside of the designated lanes must be defined. While, as a straightforward

approach, all paths are assigned an equal amount of movers, this could result in item type starvation or deadlocks if the providers have different flow rates. To avoid this, the number of movers assigned to the paths leading into each provider can be balanced according to the expected flow. While the simulation is able to create the movers in their initial positions, the physical system makes use of a MAPF solver as discussed in Section 1.1 to position the movers during initialization.

The function PLANMOVER defined in Algorithm 5 is used to perform the next action planning iteration of a mover. In order to fulfil the item sequence requirements of the receiver stations, the controller makes use of the item sequences defined in Section 2.2.4. Following the discussed strategy, the controller delays all decisions regarding sequences as long as possible, specifically to the point in time at which a mover arrives at the last awaitable node of a section. Using the action planning algorithm presented in Section 4.2, a mover continuing past this point requires an awaitable target node in a following section and therefore a commitment to one of the following sequences. For this, a depth-first recursive search is used, terminating as soon as an awaitable node is found in a valid section. If no continuation is found, the mover is considered blocked and remains at the current position until a safe tuple of actions can be generated. If multiple possible following sections exist, they are searched in an order according to the largest difference between the historic and the expected flow of the carried item type along the corresponding sections. This approach aims to match the expected flow rate, reducing the risk of starving the system of a particular item type or filling all available queue spaces along a path, which would lead to a deadlock of the system.

This planning is implemented recursively using the functions defined in Algorithm 5. The PLANACTIONSRECURSIVE function, which wraps the PLANACTIONS function of Algorithm 3, navigates to the last awaitable node of a section. If no further awaitable node exists until the end of the section, the PLANNEXTSECTION is instead called inside PLANACTIONSRECURSIVE.

PLANACTIONSRECURSIVE then computes the subset of allowed sections N_{allowed}, consisting of all outgoing sections with at least one sequence capable of accommodating the item carried by the mover. N_{allowed} is then depth-first searched for a safe continuation by calling a new instance of PLANACTIONSRECURSIVE on the chain of edges contained in the section. As soon as an awaitable node is encountered, the recursive tree is collapsed and the combined tuple of actions is performed.

Notably, this approach may result in successful navigations of a section, but an overall unsafe tuple of actions if no awaitable node can be reached. In this case, the branch is abandoned. For this reason, the commitment to all actions and sequences, such as the update of $t_{N,\text{nextFree}}$ of a traversed node N, is performed in the function PLANMOVER after a safe tuple of actions has been determined. If multiple sequences of a section are able to accommodate the mover, the controller commits to the least recently used sequence.

After each planning operation, the corresponding mover is assigned a state dependent on its outcome. In this, a mover can either be scheduled to perform an interaction, a tuple of actions, or wait for a condition before being able to continue moving. If the planning is successful in finding a safe tuple of actions, the mover is assigned the TOGOAL state. In this state, t_{nextPlan} designates the time by which the controller must have planned the next actions to prevent the mover from waiting. As the simulation does not account for deceleration time, this is equal to the finalization time of the last action $t_{\text{lastAction,end}}$, while a real-time controller would benefit by accounting for the scheduling, computation and deceleration time in order to prevent an unnecessary halting. An exception to this is if the target is a station, as in this case the TOSTATION state is applied and the simulation will initialize the station interaction after t_{nextPlan} is reached.

If the mover cannot reach the intended target node of the planning due to arriving early at a node and violating Equation 4.1, it is assigned the WAIT state if no safe actions can be performed. If instead a safe fallback node exists, the mover is assigned the TOWAIT state and travels to the fallback node before the state is converted to WAIT. Then, the mover will idle until the violating action will be considered safe upon arrival of the mover, allowing the controller to continue the pathfinding. If the mover cannot reach the intended target node of the planning and the controller has not yet assigned an action resolving the violation, no prediction is made on the time the violation is resolved. Instead, the mover is considered

blocked and a dependency to the violation is created. Consequently, the planning of the blocked mover is halted until the dependency is resolved by another successful planning. Similar to waiting states, the mover is set to the BLOCKED state if no actions were assigned, otherwise to the TOBLOCKED state, which is converted to the BLOCKED state if no new actions are planned before the completion of the final assigned action.

A mover attempting to enter a node reserved as the action target by another blocking mover is not allowed to proceed as discussed in Section 4.2, instead a BLOCKED(Mover) dependency is created from the blocked to the blocking mover. After a blocking mover is successfully scheduled new actions, all dependencies are resolved by scheduling the blocked movers to be planned next. While this is performed immediately in the simulated controller, a real-time controller must consider the increased computation time caused by the dependency releases, potentially scheduling them for a later point in time. Alternatively, a mover can be blocked from entering a section due to the item sequence assignment, creating a BLOCKED(Item) dependency from the blocked mover to the blocking section start node, which can be resolved by any mover advancing an item sequence to match the blocked mover. A mover committing to a sequence with dependencies inherits all dependencies and schedule the replanning of the blocked movers. Finally, if a mover has valid following sections, but none can be successfully navigated to a free awaitable node, the mover is assigned a BLOCKED(Node) state. The resolution of this state is more complex than other variants, requiring dependencies on all encountered blocking conditions, which must all be returned after the blocked mover has been successfully replanned. As an alternative, which is used by the implemented controller, a mover with the BLOCKED(Node) state can instead be rescheduled after a fixed time to attempt a replanning.

Algorithm 5 Full action planning algorithm of the controller.

function PLANMOVER(Mover M, computation time t_{now})
  Start time of the first action $t_{\text{start}} \leftarrow t_{\text{now}}$ ▷ In a physical system, the computation duration should be considered here.
  $N_{\text{last}} \leftarrow$ tail of last action
  $E_{\text{remaining}} \leftarrow$ tuple of edges remaining from last planning attempt
  if $E_{\text{remaining}}$ **is not** empty **then**
    $\text{actions}, \text{newState} \leftarrow$ PLANACTIONSRECURSIVE$(M, t_{\text{start}}, E_{\text{remaining}})$
  else
    $\text{actions}, \text{newState} \leftarrow$ PLANNEXTSECTION$(M, t_{\text{start}}, N_{\text{last}})$
  end if
  if actions **is not** empty **then**
    $t_{M_k,\text{nextPlan}} \leftarrow t_{\text{lastAction,end}}$
    Commit to all actions and new sections

    /* Resolve all dependencies by planning blocked movers with
     * BLOCKED(MOVER) of M and BLOCKED(ITEM) of new sections */
    $M_{\text{dependent}} \leftarrow M_{\text{dependent,mover}} \cup M_{\text{dependent,newSections}}$
    for $M_k \in M_{\text{dependent}}$ sorted by earliest $t_{\text{lastAction,end}}$ **do**
      PLANMOVER(M_k, t_{now})
    end for
  end if
end function

function PLANACTIONSRECURSIVE(Mover M, start time t_{start}, edges to plan E)
  $N_{\text{start}} \leftarrow$ first node in E
  $N_{\text{last}} \leftarrow$ last awaitable node in E
  $N_{\text{goal}} \leftarrow$ final node in E
  $\text{actions}, \text{blockingCondition} \leftarrow$ PLANACTIONS(M, t_{start}, E) ▷ Defined in Algorithm 3
  if N_{goal} is STATION and reached by actions **then**
    /* Mover travels to station for interaction. */
    $\text{newState} \leftarrow$ TOSTATION
  else if N_{last} is reached by actions and N_{last} is not N_{start} **then**
    /* Mover travels to final awaitable node. */
    remove actions after N_{last} from actions
    $\text{newState} \leftarrow$ TOGOAL
  else if N_{goal} is reached by actions **then**
    /* No later awaitable nodes available, the next section must be chosen now */
    $t_{\text{next}} \leftarrow t_{\text{lastAction,end}}$
    $\text{newActions}, \text{inheritedNewState} \leftarrow$ PLANNEXTSECTION$(M, t_{\text{next}}, N_{\text{goal}})$
    $\text{actions} \overset{\text{put}}{\leftarrow} \text{newActions}$
    $\text{newState} \leftarrow \text{inheritedNewState}$
  else if actions **is not** empty **then**
    /* M is blocked after remaining actions */
    $\text{newState} \leftarrow$ TOBLOCKED$(\text{blockingCondition})$
  else
    /* M is blocked */
    $\text{newState} \leftarrow$ BLOCKED$(\text{blockingCondition})$
  end if
end function

function PLANNEXTSECTION(Mover M, start time t_{start}, start node S)
  Outgoing sections $N_{\text{out}} \leftarrow \{ B \in \text{Graph} \mid \text{Start}(B) = G \}$
  Outgoing sequences $S_{\text{out}} \leftarrow \bigcup S_B \; \forall_B \in N_{\text{out}}$
  Allowed sections $N_{\text{allowed}} \leftarrow \{ B \in N_{\text{out}} \mid \text{Item}(M) \in I_P(B) \}$
  if N_{allowed} **is** empty **then**
    $\text{newState} \leftarrow$ BLOCKED(Item G)
    return
  end if
  for $N_k \in N_{\text{allowed}}$ **do** ▷ Sorted by priority as discussed in Section 4.3.
    $\text{newActions}, \text{inheritedNewState} \leftarrow$ PLANACTIONSRECURSIVE$(M, t_{\text{start}}, \text{Edges}(N_k))$
    if newActions **is not** empty **then**
      $\text{actions} \overset{\text{put}}{\leftarrow} \text{newActions}$
      $\text{newState} \leftarrow \text{inheritedNewState}$
      return
    end if
  end for
  $\text{newState} \leftarrow$ BLOCKED(Node G)
end function

5 Evaluation

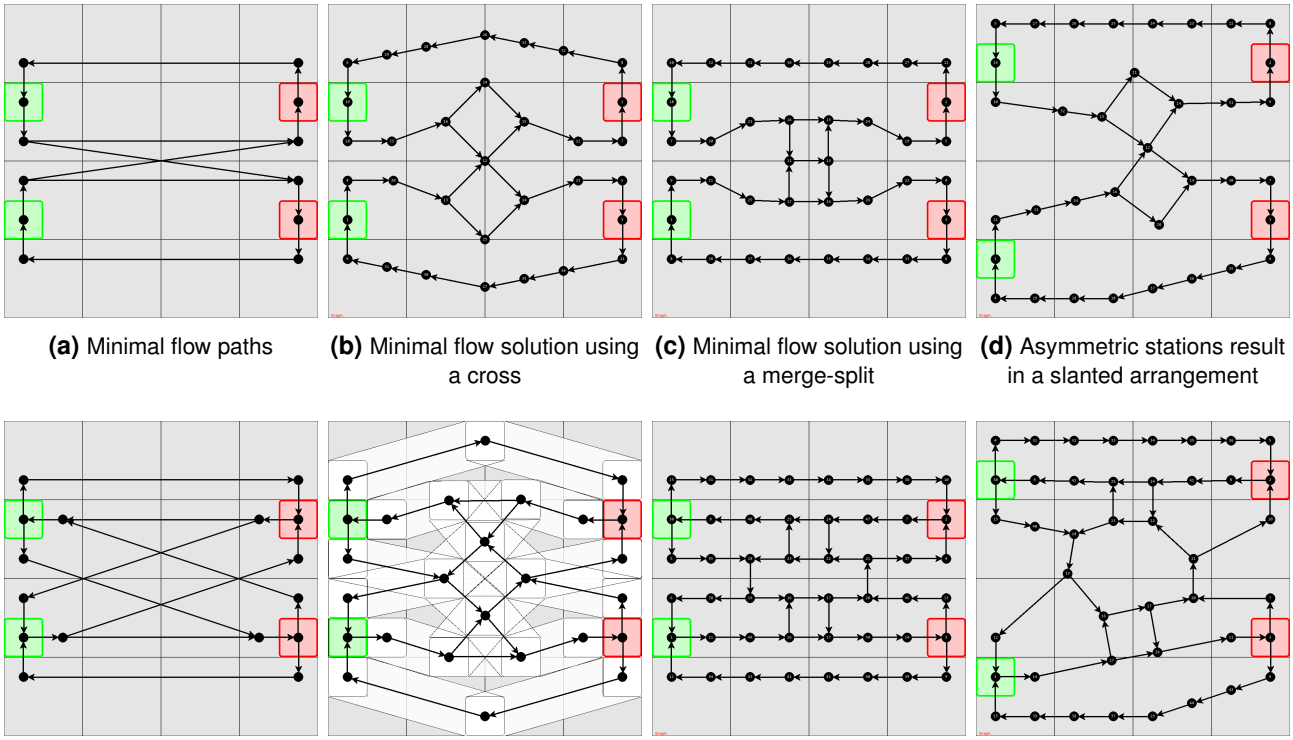
This chapter presents solutions for three environments, generated by the offline preprocessing for a number of different parameters, and discusses their validity and performance in the context of the algorithms presented in this work. Namely, these parameters are the required connectivity between stations, the number of interfaces per station and the type of intersection used. While the validity of a solution is determined according to the definition in Section 2.2.2, the performance requires the simulation of a scenario for a given number of movers. This number significantly affects the performance of a solution, as too few movers on the field cause stations to idle while too many movers result in congestions along the lanes. Due to this, a viable range is determined by performing a test series of different numbers of movers, comparing the achieved throughput as defined in Section 4.1. As the simulation does not consider acceleration and deceleration time, the assumed average velocity is set to $500\,\mathrm{mm\,s^{-1}}$, while the interaction duration τ_I has a mean of $200\,\mathrm{ms}$ and is varied by a gauss distribution using $\mu = 1$ and $\sigma = 0.1$, clamped to $[0.8, 1.2]$. Following the discussion in Section 2.3.2, this results in an expected optimal flow of $F_O = 136\,\mathrm{min^{-1}}$ for stations with opposing interfaces, while the worst-performing primitives with $90°$ corners have an optimal flow of $F_O = 125\,\mathrm{min^{-1}}$, matching the slowest possible interaction time. This choice prevents stations from having the smallest flow along their paths at all times, which could conceal the existence of bottlenecks in other areas of the graph and therefore reduce the significance of the performed tests.

While the numbers presented in this chapter do not reliably reflect the throughput of a physical application due to the assumptions taken in the mover dynamics and simulation, they are suited to compare the performance of different solutions. Furthermore, it must be considered that all presented solutions are not manually cleaned up before simulation. This results in remaining artifacts, such as slight curvatures introduced by the arc angle tolerances or missing waiting positions, which have not been removed by the automated post-processing presented in Section 3.4. While the consequences of this are further discussed in Chapter 6, the number of available waiting positions is an important aspect of the throughput of the system and will therefore affect the results presented in this chapter. In general, minor manual cleanups addressing such artifacts can further improve the performance of a system.

5.1 Simple Environment

The first exemplary environment is shown in Figure 5.1. It is populated using square primitives with an edge length of $120\,\mathrm{mm}$, placed on a square field of 8 times this unit size in length. The environment has two provider and two receiver stations with equal flow. The environment has two item types, in addition to the empty type, which are accepted by both receivers in an alternating sequence. Assuming that each provider exclusively produces one of the item types, applying the flow generation algorithm presented in Section 3.1 results in minimal flow paths as shown in Figure 5.1a, while connecting all providers with all receivers and vice-versa results in an All-to-All arrangement as shown in Figure 5.1e.

For the minimal flow paths, the pipeline generates the Solutions shown in Figures 5.1b and 5.1c for a cross and merge-split intersection respectively. As discussed in Section 2.4, the merge-split results in a more compact overall footprint, allowing a valid solution not making use of the outer edges of the field. This trend is also visible if the solution must accommodate an All-to-All flow, which significantly increases the number of intersections as can be seen in the flow paths of Figure 5.1e. A solution solely relying on cross intersections is not able to yield a valid network, with the optimizer failing to resolve the bypass arc angles at the point of the optimization shown in Figure 5.1f. The arrangement of the stations at opposing ends of the field however suits a solution using merge-split intersections, yielding a very compact result as shown in Figure 5.1g. If waiting positions are omitted, this solution only requires a 6×6 unit size footprint. This however comes at the cost of routing complexity, as the shown arrangement yields three overlapping circular arc chains, which require computationally costly cycle item sequences as discussed in Section 2.2.4.



(a) Minimal flow paths

(b) Minimal flow solution using a cross

(c) Minimal flow solution using a merge-split

(d) Asymmetric stations result in a slanted arrangement

(e) All-to-All flow paths

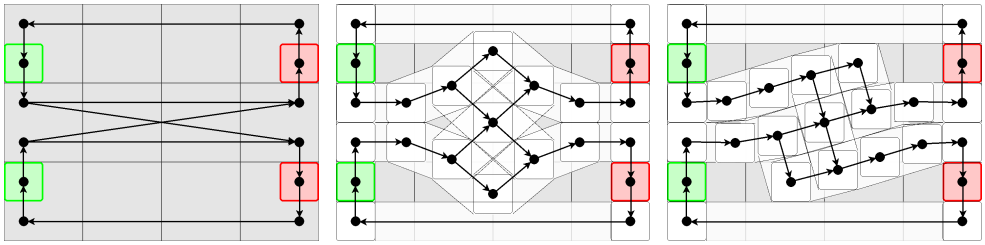
(f) Invalid All-to-All flow solution using a cross

(g) All-to-All flow solution using a merge-split

(h) Primitives remain tightly packed in their arrangements

Figure 5.1 Solutions for an environment with a 8×8 unit sizes large field for various generator configurations. For the minimal flow, both providers are connected to both receivers, while one return connection each is sufficient.

Promoting existing symmetries in an environment to be carried along has in shown in various tested environments to be beneficial for efficiency and robustness, both for the offline network optimization and online controller. During the optimization, a symmetric network typically yields very similar metrics for overlaps and violations in each of the mirrored occurrences, allowing a safe simultaneous modification of multiple points in one iteration. Furthermore, a solution resulting in similar paths between stations with equal flow simplifies the balancing required by the controller in order to prevent deadlocks and achieve equal utilization of all stations. In some cases however, sustaining the symmetry of a network is unfavorable or even preventing the convergence of the optimization. An example of this is presented in Figure 5.2, showing the environment of Figure 5.1 on a narrower field. This results in the solution shown in Figure 5.1b to no longer fit the field, leading to an unsolvable overlap shown in Figure 5.2b. While the presented optimization attempts to find a symmetric solution and therefore fails to converge, slanting the cross arrangement as shown in Figure 5.2c allows it to fit in the available space. A less severe example of symmetries negatively impacting the performance of a solution can be observed in Figure 5.1b, where both pairs of arcs incoming and outgoing from the cross arrangement bridge a distance of 1.5 index lengths. Shifting the cross arrangement to one side would result in an arrangement with 1 and 2 index length arcs respectively, allowing the placement of additional waiting spaces.



(a) Minimal flow paths on the slimmer field

(b) A symmetric cross arrangement is too wide

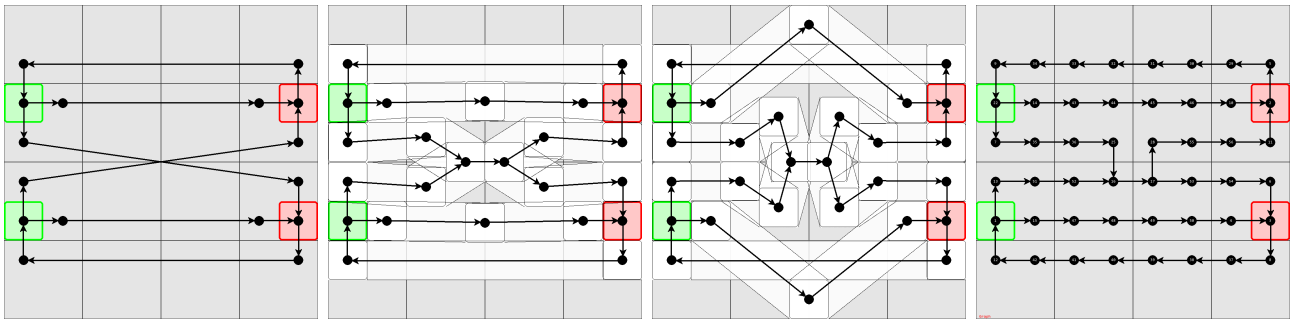
(c) A slanted cross arrangement fits the field

Figure 5.2 The environment of Figure 5.1 on a slimmer 8×6 field. The bypassed cross arrangement is slanted to generate a valid solution, requiring a breaking of the symmetry of the network.

In asymmetric environments, the continuous placement of primitives allows the optimization to compensate for them, resulting in solutions as shown in Figures 5.1d and 5.1h. In these cases, the optimization typically slants groups of primitives as a rigid unit, allowing it to maintain the relative connections between them. Examples of this are visible in both solutions with the bypassed cross of Figure 5.1d and both smaller cycles of Figure 5.1h maintaining their compact shape while being rotated to align with the fixed interfaces. This however increases the rate of occurrence of a typical shortcoming of the post processing, as can be seen along the path connecting the lower provider to the lower receiver in Figure 5.1h. In this arrangement, both incoming and outgoing arcs have a length of just under two index lengths, preventing the addition of a waiting primitive to either arc. While this could be resolved by increasing the coarseness of the position rounding of the post-processing, a too coarse grid will prevent the formation of straight lanes between the primitives, as the ideal slant can no longer be resolved. In this case, a manual intervention placing the primitives according to the analytic solution is required and has the potential to significantly increase the performance of the system.

A method to reduce the complexity of a network, further discussed in Section 2.4, is the addition of a third interface to a station. While this approach reduces the possible flow through a station, it integrates a split or merge primitive which does not need to be placed on the field. Figure 5.3a shows such an arrangement, replacing the bypasses of the intersection with two additional paths directly connecting the stations. While this reduces the number of required primitives, the additional paths result in doubled arcs lining the perimeter of the central intersection. While a straightforward solution bulging both arcs outwards would allow sufficient space for an intersection in the center of the field, this requires the network modification, following the "inside-out" approach introduced in Section 3.3.2, to split both arcs in successive iterations. If the metrics of corner addition and removal are not carefully chosen, it is possible that the

optimization is incapable of introducing a corner into the outer arc without merging the inner arcs in the same iteration. As shown in Figure 5.3b this then results in the optimization stalling. If the placement optimization forces are instead chosen such that the corner of the inner arc can pass through the outer as shown in Figure 5.3c, the resulting overlapping arcs cannot be recovered, following the discussion in Section 2.2.1. An alternative solution is found when ignoring the symmetry of the system, resulting in the configuration shown in Figure 5.3d.



(a) Minimal flow paths using three interfaces per station instead of bypasses

(b) Confined merge-split, unable to simultaneously bulge both outer arcs

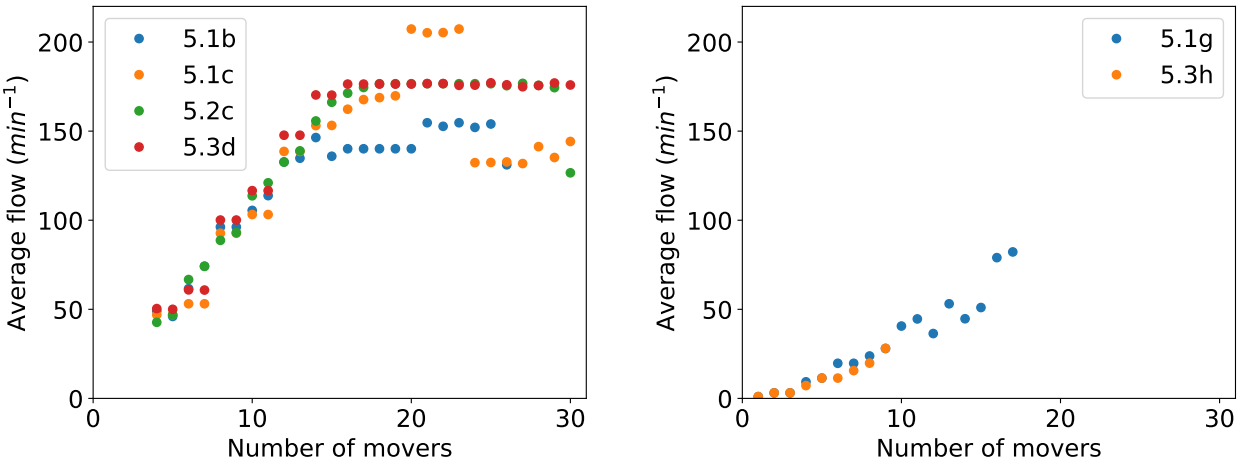
(c) Too large forces result in the corner of the inner passing through the outer arc

(d) Alternative, asymmetric solution

Figure 5.3 The environment of Figure 5.1 using a third interface instead of the bypass. This results in doubled arcs confining the center arrangement, requiring a simultaneous splitting of both arcs.

Simulating the performance of the presented valid solutions, with the results shown in Figure 5.4, yields further insight into their viability in a system expecting high throughput. The cross solution shown in Figure 5.1b achieves an average flow through both stations of $154\,\mathrm{min}^{-1}$ for ranges from 21 through 25 movers, while the merge-split in Figure 5.1c peaks at $207\,\mathrm{min}^{-1}$ for 20 to 23 movers, with a sharp drop-off to $132\,\mathrm{min}^{-1}$ at 24 movers, caused by a back up along the outer returning lanes blocking the receiver stations. In this, the latter reaches about 82% of the theoretically achievable throughput of the lanes. The observed discrepancy between the intersection types matches the expectations stated in Section 2.4, as the short batch size and long arc lengths along the cross negatively affect its performance. Slanting the cross arrangement, as is the case in Figure 5.2c, allows the intersection to partially compensate for this, reaching a flow of $176\,\mathrm{min}^{-1}$ for 17 through 29 movers. Similarly, the solution shown in Figure 5.3d making use of the third interface achieves a maximal flow for $176\,\mathrm{min}^{-1}$ for 16 to 41 movers. As expected, this value is lower due to the additional overhead introduced by using the central interface. This lower flow can already be achieved by a smaller number of movers, while the large amount of available waiting positions results in a large range of viable configurations before any of the lanes fill up and backlog the stations.

Due to the increased complexity of the sequences in the environments requiring All-to-All connectivity, the number of primitives must be reduced to prevent deadlocks occurring. This results in a significant decrease of throughput in the network of Figure 5.1g, achieving only $82\,\mathrm{min}^{-1}$ for 17 movers, with additional movers resulting in deadlocks of the system due to a bias of the return paths leading to starvation of one of the item types. The similar network of Figure 5.1h achieves a comparable flow of $89\,\mathrm{min}^{-1}$ for 22 trough 24 movers. While this could be optimized by prioritizing the same-side return paths when planning the destination stations, this requires a dedicated controller optimization outside the scope of this work.



(a) Average flow rate for environments with minimal flow path

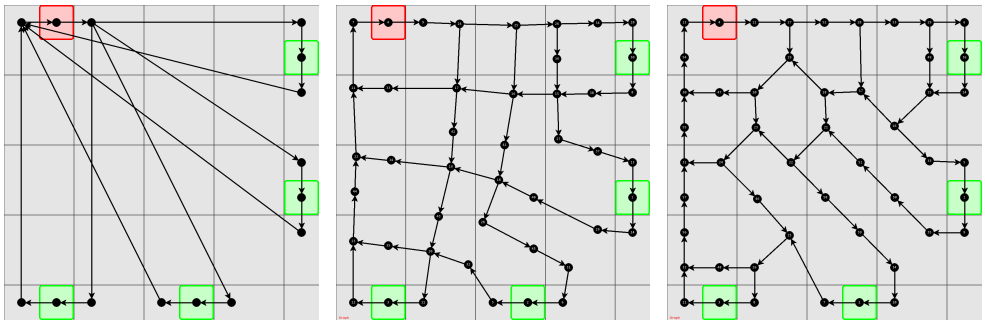
(b) Average flow rate for environments with All-to-All flow.

Figure 5.4 Average flow rates for different numbers of movers. If a deadlock has occurred for a number of movers during simulation, no value is given.

5.2 Merging Feeds Environment

The environment discussed in this section, shown in Figure 5.5, consists of four providers supplying a single receiver. While the throughput of such an arrangement is severely limited by the single receiver if all stations have the same expected flow and interaction time, it is viable for systems based around mixing a variety of different item types or performing some time-intensive loading operation at the providers. Due to this, compared to an environment as discussed in Section 5.1, the ability to achieve reliable connections is typically more relevant than achieving maximal throughput along all lanes of the system. This is compounded by the fact that in environments with a single provider or receiver, the minimal required flow paths are an All-to-All flow, as each station requires at least one incoming and outgoing connection.

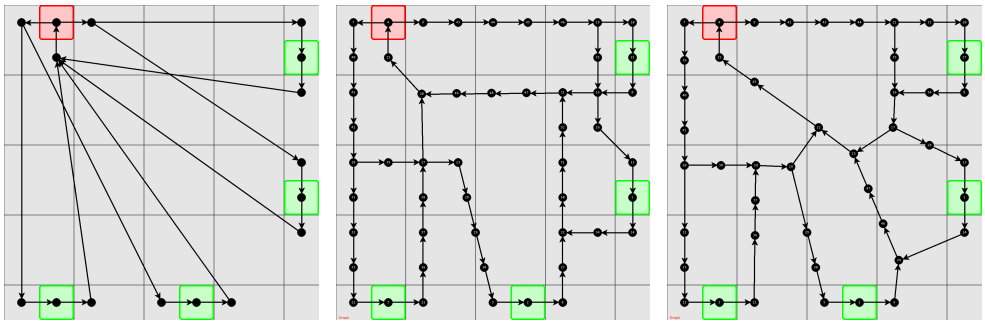
Figure 5.5a shows the flow paths generated by the offline preprocessing with each station limited to two interfaces, while the provider shown in Figure 5.5d is able to make use of a third interface, reducing the number of intersections from six to two. This is also reflected in the complexity of the presented solutions, with Figures 5.5b and 5.5c requiring a more densely populated field than Figures 5.5e and 5.5f. The network initialization implementation of the offline processing, presented in Section 3.2, is designed such that all interfaces are populated with the same corner type. While the robustness of the solver can be increased by mixing different types of intersections, the metrics required to make this decision are outside the scope of this work. Additionally, as is shown in the environments of this section and Section 5.3, large numbers of adjacent intersections can be efficiently placed when populated with the same type of primitive, resulting in quadratically or hexagonally tiled arrangements respectively.



(a) Flow using two interfaces per station

(b) Solution using two interfaces and cross intersections

(c) Solution using two interfaces and merge-split intersections



(d) Flow using a third interface for the receiver, reducing the number of intersections

(e) Solution using a third interface and cross intersections

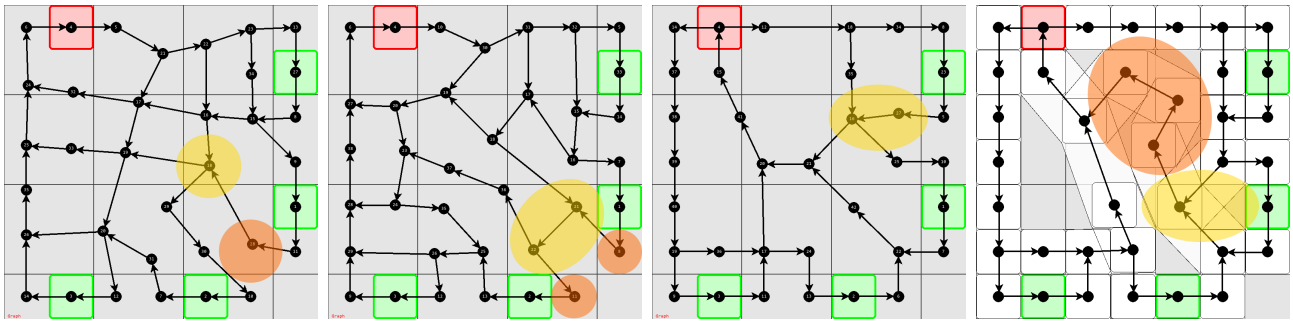
(f) Solution using a third interface and merge-split intersections

Figure 5.5 Solutions for an environment with a 9×9 unit sizes large field for various generator configurations. All four providers are connected to the receiver, and vice versa. In this, the minimal flow is an All-to-All flow.

While the arrangement of primitives on a large field such as shown in Figure 5.5 was solvable for all configurations, reducing the available field size to 7×7 unit sizes as shown in Figure 5.6 results in the optimization failing to converge. For the shown environment, the diagonally adjacent interfaces of the lower right providers prove challenging to resolve or even result in the network not having a reachable valid state from the onset. When using cross primitives as shown in Figure 5.6a, a valid arrangement requires the corner connected to the interfaces, highlighted in yellow, to be directly adjacent to the interfaces at the position highlighted in orange. If the placement optimization is not able to move the corner primitive to this position before the network modification inserts corners in an attempt to resolve the violations, the optimization is not able to recover.

If instead of the cross primitive, a merge-split is used for the intersection as is the case in Figure 5.6b, no network exists that can fulfil the validity requirements and can be reached using the available network modification options. Another example in which additional modifications of the network are required to resolve a violation is shown in Figure 5.5c, in which the cross is not able to equalize the angles between the connected arcs. A straightforward solution to this would be to swap the cross with the highlighted corner, allowing the vertical lane to run adjacent to the stations.

Figure 5.6d shows another two typical failure modes of the optimization. If the size of the field is an integer multiple of the primitive edge length, an arrangement of primitives completely filling the length is possible. If however, the step size and thresholds of the algorithm flow logic of the initial stages, discussed in Section 3.3.3, are chosen poorly, the placement optimization is not precise enough and the slotting of the final primitive does not occur before the phase is terminated. In this case, a solution slotting the merge highlighted in yellow into the gap is not found until the later stages are reached. Furthermore, if the violations encountered are sufficiently large, the network modification phase may continuously attempt to insert additional corners, leading to trails such as the one highlighted in orange. This primitive runaway typically results in bunched up, coiled arrangements with angles too large to qualify for removal.



(a) A valid solution requires the yellow cross to be placed at the orange position.

(b) The angles of the orange interfaces cannot be resolved due to the yellow merge-split.

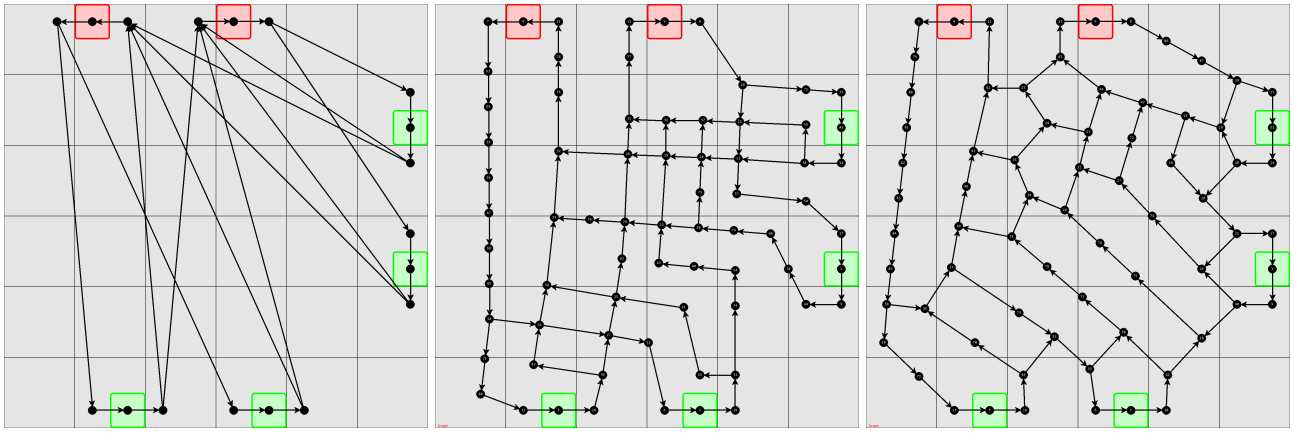
(c) The invalid angles could be resolved by swapping the yellow cross and corner.

(d) The yellow merge did not converge into the slot, resulting in a pileup of orange corners.

Figure 5.6 The environment of Figure 5.5 on a reduced 7×7 field. The interfaces of the lower right stations are too close, resulting in invalid solutions. The shaded areas highlight issues discussed in Chapter 5.

5.3 High Complexity Environment

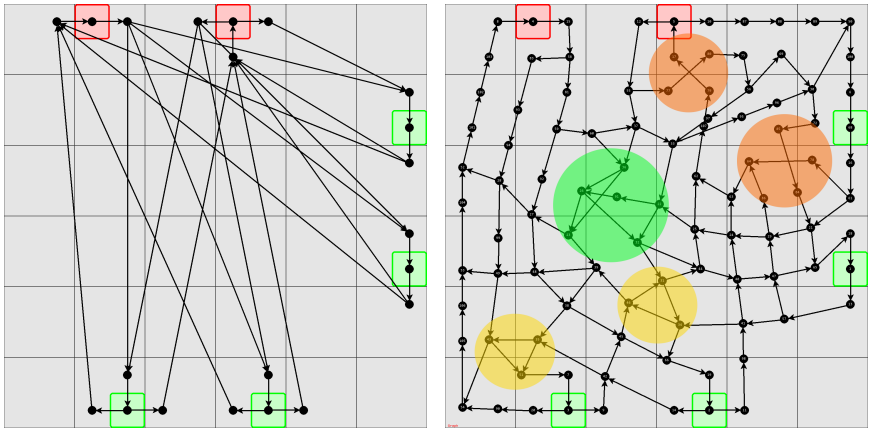
The third environment is placed on a large 12×12 unit size grid and intended to show the scale limitations of the presented offline solution generation algorithm. For this, four providers and two receivers are included in the environment, with the minimal flow path arrangement requiring each provider to be connected to all receivers, while each receiver only feeds back to two of the providers as shown in Figure 5.7a. For this flow requirement, the optimization is able to generate the valid solutions shown in Figures 5.7b and 5.7c, using cross or merge-split intersections respectively. At this scale, the advantage of cross primitives in large fields discussed in Section 2.4 becomes visible, as they are able to form a tight square grid, arranged to best match the incoming and outgoing arcs. Merge-Splits, however typically arrange themselves into cycles of five or six edges, typically arranged in a rectangular, elongated grid or a hexagonal tiling. While this increased flexibility, largely due to the $120°$ angles of the hexagonal pattern, allows for a more robust solution generation, it also results in an increased complexity of the item sequence inheritance, introduced in Section 2.2.4.



(a) Minimal flow paths, making use of two interfaces per station.

(b) Minimal flow solution using cross intersections

(c) Minimal flow solution using merge-split intersections



(d) All-to-All flow paths. The flow generation algorithm was cut off early, resulting in a suboptimal layout.

(e) Invalid solution generated by the optimization, containing a number of unrecoverable errors.

Figure 5.7 A large 12×12 unit size environment, with four providers and two receivers. The minimal flow requires all providers to be connected to both receivers, while each receiver only returns movers to two providers. The presented optimization is capable of generating valid solutions for the minimal flow, but fails at the All-to-All flow. The shaded areas highlight issues discussed in Chapter 5.

If, instead of the minimal flow, an All-to-All flow is required for the environment, the number of intersections and resulting complexity increases drastically. To simulate a more reasonable sub-optimal approach to the flow generation assignment of Section 3.1.3, the flow paths shown in Figure 5.7d have been generated only testing a subset of possible combinations of paths and interface combinations. This has resulted in an suboptimal layout with 36 intersections, instead of the minimally achievable 30. Despite this, this is a significant increase from the 10 intersections of the minimal flow shown in Figure 5.7a and results in the optimization failing generate a valid network. Figure 5.7e shows a snapshot of the optimization, highlighting some of the significant encountered violations.

While the number of cross primitives required to populate all intersections only make up a small portion of the overall available space, the layout of the environment results in a high initial concentration of intersections in a small area. While the step size is decreased during the first steps of the placement optimization as discussed in Section 3.3.1, this is not sufficient for the intersections to distribute. In this case, there is a high likelihood that some of the overlaps are resolved in a direction leading to additional overlaps of arcs and therefore unrecoverable violations as highlighted in orange. Futhermore, the design of the network modification phase is such that only a single violation, or a set of equal violations, can be resolved in each iteration. While this is necessary to prevent the optimization from making redundant modifications while attempting to resolve a single violation, this results in a slow convergence for large environments with many independent violations, such as the triangles highlighted in yellow. Finally the area highlighted in green contains a bypassed cross arrangement that, due to a lack of space, was unable to properly form, resulting in an overlap of multiple arcs. In order to improve the robustness of the solution, other arrangements with the same connectivity, such as merge-splits, fully connected crosses, or cycles, could be employed to reduce the number of required primitives.

6 Conclusion and Outlook

This work presents an approach for efficiently controlling a large number of AGV movers in an intralogistics environment. For this, a preprocessing pipeline has been introduced, autonomously generating a solution graph for a given environment. In this, a novel item sequence method is proposed to efficiently ensure a valid unloading order of the transported item types. Furthermore, a controller design capable of operating on low-performance, real-time PLCs has been presented, and shown to be able to efficiently control fleets of movers in a simulated environment based on the generated solution.

The problem statement in Chapter 2 defines a number of requirements to such a system, which have been fulfilled as follows. The system is intended to be operated on a planar motor system as part of a larger assembly, with the purpose of rearranging products while transporting them from a preceding to a following subassembly. This has been considered in the design of the underlying models of the approach, by analyzing the capabilities and dynamics of such systems, and by defining interaction stations providing an interface suitable for operation within typical industrial assemblies. Furthermore, the system is required to operate within the limits of IEC 61131 compliant PLCs, ensuring a reliable and deadlock-free operation and a minimum sustained throughput while abiding to the given item type sequences. While the operation of such a system has only been shown in simulation, the underlying algorithm is designed such that an implementation using IEC 61131-3 is feasible. In this, the proposed item sequences have been shown to guarantee the adherence to the provided station sequences, while a sufficient throughput and a deadlock-free operation have been determined with reasonable confidence using a series of endurance tests. Finally, a suitable solution for an environment must be computed efficiently for various configurations, while adhering to additional specifications such as path redundancy, robustness to variances in timings and limited station buffer sizes. The examples provided throughout this work, specifically in Chapter 5, demonstrate the ability of the offline preprocessing to generate solutions to a wide variety of configurations. Furthermore, this works highlights areas in which the proposed methods have been insufficient and proposes alternative approaches and solutions to accommodate such environments.

Arguably the largest limitations posed on the algorithm are found in the layout generation. The requirement of a convex field, without obstructions and with all stations placed at the field border are severely restricting the environments which can be solved by the presented algorithm. To some of these restrictions, straightforward solutions exist. As the convexity is merely required to ensure the completeness of the intersection detection introduced in Section 3.1, any field can be included which can be topologically consistently transformed into a field with a convex border. Then, the required intersection detection can be performed on the transformed field, as all intersections are maintained. While this resolves all non-convex fields, this approach does not resolve obstacles or stations placed away from the border. In this case, each path may be routed along either side of the obstacle, severely increasing the already exponentially growing space of possible matchings to be searched. While non-optimal methods such as simulated annealing [47] may be used to more efficiently search viable matchings, the exponential growth of the search space is a limiting factor to the performance of the presented algorithm. Due to this, alternative pathfinding methods should be considered, which are already commonplace in routing tasks on higher performance systems. As this task is performed in the preprocessing phase, no requirements to real-time exist and the full capabilities of modern workstations are available. In this case, a dense, fully connected bidirectional grid can be searched until a suitable arrangement of paths is found, which then represents the initial layout for the following operations in the pipeline.

The evaluation examples presented in Chapter 5 have revealed a number of shortcomings of the optimization algorithm. While a number of them are specific to the environment, requiring either further tuning of the optimization parameters or a manual resolution, others can efficiently be addressed by modifying the optimization. A straightforward addition is to include a detection of overlapping arcs which are, as discussed in Section 3.3.1, redundant to the detection of primitive-arc collisions for already planar networks and therefore omitted to improve performance. Nevertheless, intermittently performing this test as a sanity check to ensure that the network is still planar may prove useful in determining if the optimization is still stable or has devolved into an otherwise unrecoverable state. Throughout the examples presented in this work, arcs are assumed to have a minimum length of one index length. As discussed in Section 2.2.2 however, arcs can benefit from changing this length, either by allowing the overlap of connected primitives or by enforcing a minimal number of waiting spaces. While an automatic post-processing was performed on all examples shown in the evaluation, most of the more complex arrangements have remaining artifacts of the optimization. Specifically, this involves a slight curvature introduced by the arc angle tolerances of the finalization stage, and arc lengths just below the next integer multiple of the index length. While a bias is introduced in the arc step optimization component discussed in Section 3.3.1, this is not always sufficient, resulting in missed waiting positions and overly long arc. While an increased coarseness of the position rounding applied during the post-processing is somewhat able to address this, a more extensive approach, possibly orienting aligned nodes at an average formed over their neighborhood, is required to reduce the need to manually post-process the resulting solutions.

Another potentially useful primitive type is an All-to-All cross, allowing a continuation along both outgoing arcs independent of the incoming arc. While such a primitive would then serve as a section start and goal and, as discussed in Section 2.4, increase the complexity of the control, it serves as the smallest size intersection allowing same-side connectivity. Especially in complex, but low throughput environments such as shown in Figure 5.7e, replacing the bypassed cross highlighted in green with a single primitive is likely to improve the robustness of the system. As a simplification, the network initialization presented in Section 3.2 makes exclusive use of a single type of intersection. However, as discussed in Section 2.4, each type has its strengths and weaknesses, leading to the conclusion that mixing them can prove beneficial to the performance of the optimization. The implementation of this however requires either manual intervention, or the introduction of some metrics to determine the ideal type for each occurrence.

Figure 5.5c of the evaluation shows an environment, in which the highlighted corner cannot achieve a viable configuration, as it is blocked by an adjacent corner primitive. While, as a minimal working example, the current network modification phase introduced in Section 3.3.2 is only capable of removing or adding corners, additional types of modifications, such as swapping two connected and adjacent primitives, have the potential to greatly improve the performance of the system. Such an operation however is a non-continuous movement and must therefore be thoroughly checked for planarity and topological consistency of the modified network before it is committed. A similar extension to the network modification algorithm concerns itself with merge or split primitives. While, in the interest of increasing the flexibility of the angles of the attached arcs, only two-way splits and merges are currently implemented, systems may benefit from relaxing this and instead merging two connected and adjacent primitives of the same type in order to create a three-way split with a reduced footprint and improved symmetry properties.

As shown in the example of Section 5.3, large solutions with many independent violations tend to converge slowly, as only one violation is addressed at a time, barring symmetries. This measure is taken to prevent the application of multiple redundant modifications addressing the same violation. This process could be accelerated drastically by showing the independence of two violations with reasonable confidence, allowing the application of multiple modifications in the same iteration. A straightforward approach to this is the detection of arrangements which are guaranteed to require a modification, such as shown in Figure 3.10 of Section 3.3.2. For such arrangements it is reasonable to presume that, after the initial settling of the network, a split according to the highest corresponding score of the contained arcs is best suited to resolve it, allowing multiple such violations to be addressed simultaneously.

While the simulation presented in Chapter 4 agrees with the analytical observations throughout this work, a number of assumptions and simplifications have been taken. As the evaluation of the resulting differences to a real system are outside the scope of this work, additional research is required to determine a balance between computational efficiency and accuracy, allowing the verification of a generated solution in software before employing it on live hardware. Most significant in this are the simplifications to the mover dynamics, estimating the differences in timing due to acceleration and precise collision detection. Furthermore, the performance of the movers is assumed independent of the loaded products, whose mass may significantly affect the achievable acceleration and top speed. While the presented real-time controller has been shown to operate reliably on a number of solutions, some configurations presented in Chapter 5 have shown to perform poorly due to the routing decisions of the controller. Especially more complex arrangements with redundant paths and high throughput have resulted in deadlocks for the number of movers required to fully saturate the stations. In this, additional work developing novel decision algorithms based on the presented item sequences is necessary to improve the robustness and performance of the controller on such solutions.

The presented preprocessing is designed with supervised operation in mind. In this, the proposed algorithm is used as a supporting tool aiding an expert in laying out a system. This is further reinforced by the modular design, as each step of the pipeline may be used individually, allowing a supervised operation to work around some of the limitations of the presented algorithm, for example by applying the placement optimization to a hand-crafted network for an environment too complex to be solved satisfyingly by the flow generation. While networks are designed to be represented in a compact text-based format, this process can be further improved by providing an interactive GUI, allowing a simultaneous supervision of the layout and optimization process while enabling the operator to intervene at any time by halting the processing and modifying the network as necessary.

During surveys with industry partners, a number of requirements have been formulated which have been placed outside the scope of this work. Namely, these are the introduction of rotation points and more complex station types. Currently, all primitives and movers are axis-aligned at all times. The employed planar motor systems however are capable of rotating movers in increments of $90°$. Making use of this feature would, in addition to the transporting, collecting and grouping tasks mentioned in Chapter 1, allow the system to also perform the orientation of the products. As this ability however requires a specific placement of the movers on the tiles, the autonomous positioning of such primitives on the field is not straightforward. The second addition requested by the industry partners concerns itself with the coordination of stations. Using traditional hardware, multiple items are frequently moved simultaneously, after being grouped in a specific configuration. The current model of the station is defined such that they interact with a single mover at a time and are independent of each other. A relaxation of this specification, allowing multiple movers to enter and interact with the station simultaneously, would allow the inclusion of such use cases. Futhermore, as discussed in Sections 3.1.2 and 5.2, station interfaces are not shared between stations and are, if placed too closely, a likely source of failure of the optimization. Relaxing the requirement and allowing interfaces to serve as splits or merges, connecting multiple arcs or stations, is likely to lessen this risk. As discussed in the problem statement in Chapter 2, stations are considered fixed to allow a change of the configuration without needing to modify the hardware. During the initial setup of a system however, stations are typically able to be moved to some degree, such as along a section of the field border. In this case, allowing the stations with attached interfaces to be moved during placement optimization is likely to result in superior solutions.

Bibliography

[1] Stefan Biffl, Arndt Lüder, and Detlef Gerhard, editors. *Multi-Disciplinary Engineering for Cyber-Physical Production Systems*. Springer International Publishing, Cham, 2017.

[2] Yonggui Wang, Jongkuk Lee, Er (Eric) Fang, and Shuang Ma. Project Customization and the Supplier Revenue–Cost Dilemmas: The Critical Roles of Supplier–Customer Coordination. *Journal of Marketing*, 81(1):136–154, January 2017.

[3] Florian Butollo and Lea Schneidemesser. Beyond “Industry 4.0": B2B factory networks as an alternative path towards the digital transformation of manufacturing and work. *International Labour Review*, 160(4):537–552, December 2021.

[4] Abdelhak Dahmani, Lyes Benyoucef, and Jean-Marc Mercantini. Toward Sustainable Reconfigurable Manufacturing Systems (SRMS): Past, Present, and Future. *Procedia Computer Science*, 200:1605–1614, 2022.

[5] Stephan Kessler and Ludger Brüll. New Production Concepts for the Process Industry Require Modular Logistics Solutions. *ChemBioEng Reviews*, 2(5):351–355, 2015.

[6] Johannes Fottner, Dana Clauer, Fabian Hormes, Michael Freitag, Thies Beinke, Ludger Overmeyer, Simon Nicolas Gottwald, Ralf Elbert, Tessa Sarnow, Thorsten Schmidt, Karl Benedikt Reith, Hartmut Zedek, and Franziska Thomas. 2.1.2 Actuators. In *Autonomous Systems in Intralogistics – State of the Art and Future Research Challenges*, page 8. Bundesvereinigung Logistik (BVL) e.V., DE, 2 edition, February 2021.

[7] M. De Ryck, M. Versteyhe, and F. Debrouwere. Automated guided vehicle systems, state-of-the-art control algorithms and techniques. *Journal of Manufacturing Systems*, 54:152–173, January 2020.

[8] Michelle Blumenstein, Vincent Henkel, Alexander Fay, Andreas Stutz, Stephan Scheuren, and Niklas Austermann. Integration of Flexible Transport Systems into Modular Production-Related Logistics Areas. In *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*, pages 1–8, July 2023.

[9] Planar Motor Incorporated. Planar Motor. https://planarmotor.com/en, November 2024.

[10] B&R Industrial Automation GmbH. ACOPOS 6D. https://www.br-automation.com/en/products/mechatronic-systems/acopos-6d/, November 2024.

[11] Beckhoff Automation GmbH & Co KG. XPlanar. https://www.beckhoff.com/en-en/products/motion/xplanar-planar-motor-system/, November 2024.

[12] SOMIC Verpackungsmaschinen and GmbH & Co. KG. SOMIC CORAS. https://www.somic-packaging.com/en/solutions/collating-and-grouping-systems/coras.html, November 2024.

[13] Beckhoff Automation GmbH & Co KG. Provisur FMS. https://www.beckhoff.com/en-en/company/news/floating-product-transport-cuts-footprint-in-half-and-speeds-up-handling-and-cleaning.html, November 2024.

[14] SPS-MAGAZIN. XPlanar. https://www.sps-magazin.de/steuerungstechnik-sps-ipc-cnc/xplanar-flying-motion/, November 2024.

[15] packaging journal. Endverpacken – zukunftsorientiert und innovativ. https://packaging-journal.de/endverpacken-zukunftsorientiert-und-innovativ/, November 2024.

[16] Ian Clark and Sias Mostert. IS THERE A PLC ALTERNATIVE FOR INDUSTRY? *IFAC Proceedings Volumes*, 39(17):125–130, 2006.

[17] Reinhard Langmann and Michael Stiller. The PLC as a Smart Service in Industry 4.0 Production Systems. *Applied Sciences*, 9(18):3815, September 2019.

[18] László Monostori. Cyber-physical Production Systems: Roots, Expectations and R&D Challenges. *Procedia CIRP*, 17:9–13, 2014.

[19] Sebastian Thiede. Cyber-Physical Production Systems (CPPS): Introduction. *JMMP*, 5(1):24, March 2021.

[20] Birgit Vogel-Heuser, Susanne Rösch, Juliane Fischer, Thomas Simon, Sebastian Ulewicz, and Jens Folmer. Fault Handling in PLC-Based Industry 4.0 Automated Production Systems as a Basis for Restart and Self-Configuration and Its Evaluation. *JSEA*, 09(01):1–43, 2016.

[21] Fazel Ansari, Robert Glawar, and Wilfried Sihn. Prescriptive Maintenance of CPPS by Integrating Multimodal Data with Dynamic Bayesian Networks. In Jürgen Beyerer, Alexander Maier, and Oliver Niggemann, editors, *Machine Learning for Cyber Physical Systems*, pages 1–8, Berlin, Heidelberg, 2020. Springer.

[22] Birgit Vogel-Heuser, Alexander Fay, Ina Schaefer, and Matthias Tichy. Evolution of software in automated production systems: Challenges and research directions. *Journal of Systems and Software*, 110:54–84, December 2015.

[23] Eva-Maria Neumann, Birgit Vogel-Heuser, Juliane Fischer, Felix Ocker, Sebastian Diehm, and Michael Schwarz. Formalization of Design Patterns and Their Automatic Identification in PLC Software for Architecture Assessment. *IFAC-PapersOnLine*, 53(2):7819–7826, 2020.

[24] Birgit Vogel-Heuser, Juliane Fischer, Dieter Hess, Eva-Maria Neumann, and Marcus Würr. Boosting Extra-Functional Code Reusability in Cyber-Physical Production Systems: The Error Handling Case Study. *IEEE Transactions on Emerging Topics in Computing*, 10(1):60–73, January 2022.

[25] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 391–400, September 2014.

[26] Bernhard Werner. Object-oriented extensions for iec 61131-3. *IEEE Industrial Electronics Magazine*, 3(4):36–39, December 2009.

[27] Ricardo Patricio and Abel Mendes. Consumption Patterns and the Advent of Automated Guided Vehicles, and the Trends for Automated Guided Vehicles. *Curr Robot Rep*, 1(3):145–149, September 2020.

[28] Bozer Yavuz Ahmet and Mandyam M. Srinivasan. Tandem configurations for automated guided vehicle systems and the analysis of single vehicle loops. Technical report, 1988.

[29] Gunter P. Sharp and Fuh-Hwa Franklin Liu. An analytical method for configuring fixed-path, closed-loop material handling systems†. *International Journal of Production Research*, 28(4):757–783, April 1990.

[30] Jonas Stenzel and Lea Schmitz. Automated Roadmap Graph Creation and MAPF Benchmarking for Large AGV Fleets. In *2022 8th International Conference on Automation, Robotics and Applications (ICARA)*, pages 146–153, Prague, Czech Republic, February 2022. IEEE.

[31] David Silver. Cooperative Pathfinding. *AIIDE*, 1(1):117–122, 2005.

[32] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[33] Mike Phillips and Maxim Likhachev. SIPP: Safe interval path planning for dynamic environments. In *2011 IEEE International Conference on Robotics and Automation*, pages 5628–5635, May 2011.

[34] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Conflict-Based Search For Optimal Multi-Agent Path Finding. *Twenty-Sixth AAAI Conference on Artificial Intelligence*, July 2012.

[35] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Meta-Agent Conflict-Based Search For Optimal Multi-Agent Path Finding. *SOCS*, 3(1):97–104, July 2012.

[36] Thayne T. Walker, Nathan R. Sturtevant, and Ariel Felner. Extended Increasing Cost Tree Search for Non-Unit Cost Domains. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 534–540, Stockholm, Sweden, July 2018. International Joint Conferences on Artificial Intelligence Organization.

[37] Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks, May 2017.

[38] J. Cagan, K. Shimada, and S. Yin. A survey of computational approaches to three-dimensional layout problems. *Computer-Aided Design*, 34(8):597–611, July 2002.

[39] Kwang Jae Lee. *Optimal System Design with Geometric Considerations*. PhD thesis, University of Michigan, 2014.

[40] Satya R. T. Peddada, Kai A. James, and James T. Allison. A Novel Two-Stage Design Framework for Two-Dimensional Spatial Packing of Interconnected Components. *Journal of Mechanical Design*, 143(3):031706, March 2021.

[41] Satya R T Peddada, Samanta B Rodriguez, Kai A James, and James T Allison. Automated Layout Generation Methods for 2D Spatial Packing. August 2020.

[42] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[43] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw Pract Exp*, 21(11):1129–1164, November 1991.

[44] Planar Motor Incorporated. Series 3 Operating Instructions. https://www.planarmotor.com/downloads/Planar_Motor_Solution.pdf, November 2024.

[45] Beckhoff Automation GmbH & Co KG. XPlanar Operating Instructions. https://download.beckhoff.com/download/document/motion/xplanar_ba_en.pdf, November 2024.

[46] Planar Motor Incorporated. XBot M3-06. https://www.planarmotor.com/en/m3-xbot, November 2024.

[47] Peter J. M. Van Laarhoven and Emile H. L. Aarts. *Simulated Annealing: Theory and Applications*. Springer Netherlands, Dordrecht, 1987.

[48] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. August 1987.

[49] Python Software Foundation. Python. https://www.python.org/, November 2024.

[50] Planar Motor Incorporated. Planar Motor Controller. https://www.planarmotor.com/en/pmc, November 2024.

List of Figures

List of Algorithms