Joachim Dunkel, Bsc

Building a modern MAPD Routing System for Large Agents on Clothoid Curve Roadmaps

Master's Thesis

to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor
Ass. Prof. Dipl-Ing. Dr. techn. Christian Steger

Advisors
DDipl.-Ing. Dr. techn. Thomas Ulz (KNAPP Industry Solutions GmbH)
Dipl.-Ing. Christof Schützenhöfer (KNAPP Industry Solutions GmbH)

Institute of Technical Informatics

Graz, September 2024

Affidavit

I declare that I have authored this Master's thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sourced used. The text document uploaded to TUGRAZONLINE is identical to the present Master's thesis.

(Date) (Joachim Dunkel, Bsc)

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to all those who have supported and inspired me during my master's thesis. First of all, I would like to thank my supervisor, Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger, for the excellent supervision and cooperation. Your professional competence and constructive criticism have contributed significantly to the successful completion of this thesis. I would also like to thank Knapp Industry Solutions GmbH, especially my advisors and contacts, DDipl.-Ing. Dr. techn. Thomas Ulz and his team. Their guidance and support during my work at the company has enriched my research work and provided me with valuable insights into the industry. I would especially like to thank Knapp Industry Solutions GmbH for generously providing the necessary resources for my research and for the unique opportunity to work at the company during my master's thesis. Finally, I would also like to thank my family and my girlfriend for always supporting me during this journey. Their understanding, patience and encouragement have helped me to overcome this challenge.

Abstract

Efficiently solving the Multi-Agent Pickup and Delivery (MAPD) problem is a key challenge in modern logistics operations, where the need for optimal coordination of multiple agents is paramount. Traditionally used decentralized strategies, where each robot independently plans its path, often do not scale well and can result in suboptimal routing decisions. Moreover, classical Multi-Agent Path Finding (MAPF) formulations are based on grid-based environments, which simplify the problem but fail to capture the complexities of real-world scenarios, particularly when agents must navigate over continuous Euclidean roadmaps.

This thesis presents a novel Multi-Agent Pickup and Delivery (MAPD) routing system for large agents on clothoid curve roadmaps. The core of the system leverages recent advancements in Multi-Agent Path Finding (MAPF). Specifically combining the idea of pre-computing continous time conflicts with the Rolling Horizon Collision-Resolution and Execution (RHCRE) framework. To validate the proposed system, a bare-boned simulation of the existing Fleet Control System (FCS) was developed, enabling a qualitative comparison with its current decentralized routing strategy. The results indicate that the proposed system not only better avoids deadlocks and increases throughput but also significantly decreases the manual setup effort during Lanemap construction.

Table of Content

List of Figures

List of Algorithms

1 Introduction

As logistics companies expand their warehouses and face increased throughput demands, they are increasingly turning to advanced automation to efficiently manage the escalating workload. Knapp Industry Solutions (KIN) offers a diverse range of products tailored to various applications, including Autonomous Mobile Robots (AMRs) designed for autonomous transportation of goods. Employing multiple such AMRs (ie. shuttles) in a shared warehouse space allows for the automation of previously manually performed tasks like fetching palette carried goods with a forklift. To manage the coordination of such warehouse-fleets, KIN uses its own Fleet Control System (FCS). This so called FCS is the central communication and coordination platform for all AMRs in a customer warehouse. Its responsibilities involve order management, fleet coordination, and task assignment to the individual AMRs. During operation, the FCS continuously monitors the status of all AMRs and ensures their efficient movement through the warehouse environment. One crucial task for the FCS is ,therefore, to coordinate shuttle positions on this roadmap graph in a way that maximizes throughput while still being robust to commotions or deadlock scenarios. This thesis describes the development of a modern state-of-the-art routing system that incorporates recent progress on the Multi-Agent Pickup and Delivery (MAPD) problem into a comprehensive framework, to replace the fleet-coordination strategy currently used in FCS in the future.

1.1 Motivation

After a warehouse area is mapped and a 2D roadmap graph (lane map) is manually constructed, the FCS undertakes the critical task of ensuring smooth, collision free coordination of shuttles. In densely populated warehouse environments, efficient coordination (routing) becomes crucial to maintain throughput without sacrificing safety.

Currently, the FCS manages its fleets in a decentralized manner, assigning each shuttle the shortest path to its destination, without regard to other shuttles. While simple and robust, the effectiveness of this approach declines as the number of shuttles increases. Independent path-planning may lead to sub-optimal routing decisions, creating bottlenecks and potentially even deadlocks. Such issues can disrupt operational flow and degrade overall warehouse system performance. A deadlock can occur (e.g) if two shuttles drive towards each other in a narrow corridor where turning around or driving backwards is not possible.

Centralized planning, on the other hand, can calculate paths for all shuttles in such a way that they do not interfere with each other. This approach leads to more efficient routing decisions. However, it was traditionally considered too slow for real-time operations.

This thesis presents a modern routing system that incorporates recent progress on the Multi-Agent Path Finding (MAPF) problem and its extension, the Multi-Agent Pickup and Delivery (MAPD) problem into a centralized framework, This framework is directly applicable to the

routing challenges faced by the FCS, where lifelong, real-time coordination of large agents traveling over 2D clothoid curve roadmaps is required.

1.2 Objective

In this thesis, a centralized warehouse-routing system is developed. It is specifically designed to efficiently coordinate a fleet of autonomous shuttles navigating over 2D roadmap graphs, as typically utilized in FCS. The system aims to optimize route planning and shuttle deployment in real-time, ensuring maximum throughput and minimal traffic congestion within a dynamic warehouse environment. Addressing this goal involves tackling several key challenges:

- **Scalability**: The proposed routing system seeks to address scalability issues inherent in traditionally decentralized models. By centralizing decision-making, the system is expected to better manage congestion and reduce the occurrence of deadlocks, thereby enhancing the ability to scale up the amount of simultaneously operating shuttles without the corresponding throughput degeneration.
- **Resilience to Delays**: Real-world operations of AMRs frequently encounter delays due to mechanical issues, varying task execution times, or the need to avoid obstacles such as people or other shuttles. The proposed system must therefore be robust to such execution delays, capable of adapting to and compensating for these disruptions to maintain operational flow.
- **Operational Continuity**: In an industrial warehouse setting orders arrive continuously, at any time of the day. The proposed routing system must efficiently incorporate such real-world demands already present in the FCS system. It must therefore be capable of handling new orders in an online lifelong manner, dynamically adjusting the fleet's routing decisions in real-time.

1.3 Outline

Chapter 2 (Background and Related Work) provides an overview of the current state of the art in (roadmap based) warehouse fleet coordination and the existing literature on the topic. Specifically, after introducing theoretical definitions, it discusses the MAPF problem and by extension the MAPD frameworks, which form the foundation of the research presented in this thesis.

Chapter 3 (Fleet Control System (FCS)) offers an overview of the current FCS system. First the terminology and concepts used in the FCS system are introduced. Then the currently used fleet coordination strategy is described. Further, its drawbacks and existing mitigation-strategies are discussed.

Chapter 4 (Design) presents the proposed routing system. It provides a detailed description of its architecture and discusses design choices in detail. As the core of the thesis, this section specifically goes into detail on how state-of-the-art algorithms were chosen and adapted into a comprehensive framework to use for warehouse routing on arbitrary roadmap graphs.

Chapter 5 (Implementation) delves into details of the proposed routing system. A special emphasis is put on how the framework discussed in Chapter 4 (Design) was implemented and how it can be used in practice. For the evaluation a bare-boned simulation Fleet Control System - Fake (FCS Fake) was implemented. This section highlights its differences from the real FCS system and details the data export process with the actual FCS.

Chapter 6 (Evaluation) presents the evaluation of the proposed routing system. Were possible, comparisons are made to the current routing system used in FCS. A comprehensive analysis of the systems strengths and shortcomings are provided. Since MAPD is an ongoing research topic, an outlook for possible future research directions is given.

Chapter 7 (Conclusion & Future Work) summarizes the thesis's main contributions, addresses its limitations, and outlines potential future research directions.

2 Background and Related Work

This chapter provides an overview of the foundational concepts and existing research that underpin this thesis. First, the basics of graph theory essential for understanding pathfinding and routing are discussed in Chapter 2.1 (Graph Theory Concepts). Next, in Chapter 2.2 (Single Agent Path Finding (SAPF)) the basic algorithms and techniques used in Single-Agent Path Finding (SAPF) are described, followed by more advanced methods like SIPP. In Chapter 2.3 (Multi Agent Path Finding (MAPF)), theoretical concepts and definitions related to MAPF are introduced. Given the fundamental importance of the MAPF, this is followed by Chapter 2.4 (Algorithmic Strategies for MAPF), which discusses existing algorithms and solution strategies for MAPF. Chapter 2.5 (Multi-Agent Pickup and Delivery (MAPD)) introduces the MAPD problem, which extends MAPF by more accurately reflecting lifelong warehouse operation scenarios. In Chapter 2.6 (Bridging the Gap between Theory and Practice), additional real-world constraints not present in the MAPD formulation are addressed and the RHCRE framework that is central to this thesis is introduced.

2.1 Graph Theory Concepts

This chapter provides definitions and brief discussions of the graph theory concepts that are fundamental to all further topics discussed in this thesis. For a more in-depth discussion, the reader is referred to (Diestel, 2017)

2.1.1 Graph

A **graph** is a mathematical structure used to model pairwise relations between objects. It is made up of vertices (also referred to as nodes or points in this document) which are connected by edges. Formally, a graph is an ordered pair $G = (V, E)$, comprising:

- V, a set of **vertices**
- $E \subseteq \{\{x, y\} \mid x, y \in V \land x \neq y\}$, a set of **edges**, which are pairs of vertices, representing their connection.

A **directed graph**, also called digraph, is a graph where all edges have a direction, indicating a one-way relationship between two vertices. More formally an edge $e_i \in E$ represents a connection between two vertices v_i and v_j where v_i is the **source** and v_j is the **target** of the edge. In contrast, an **undirected graph** has edges without direction, showing a two-way relationship between two vertices. More formally edge $e_i \in E$ represents a connection between two vertices v_i and v_j where v_i and v_j are both **source** and **target**. An undirected graph is therefore a special case of a directed graph where, for every connected pair of vertices v_i and v_j, there are always edges in both directions $(v_i \to v_j)$ and $(v_j \to v_i)$.

2.1.2 Weighted graphs

A **weighted graph** is a graph in which each edge $e \in E$ is assigned a numerical label or weight $w(e)$. Formally, a weighted graph is represented as $G = (V, E, w)$ where $w : E \mapsto \mathbb{R}$ maps each edge to a real number. Depending on the context, weights might represent costs, lengths or needed traversal time.

2.1.3 Path

A **path** in graph theory is a sequence of vertices directly connected by edges.
Consider a simple graph with $V = \{A, B, C\}$ and $E = \{(A, B), (B, C)\}$. A path $\pi_{(A,C)}$ from A to C would be the sequence of connected vertices $A \to B \to C$, or $\pi_{(A,C)} = (A, B, C)$.
In the most formal definitions in the literature a path requires all vertices to be unique. Paths that contain the same vertex multiple times are called **walks**. However some definitions allow for paths to contain the same vertex multiple times. In this thesis, paths and walks are used interchangeably, as vertex repetition is not a concern.

2.1.4 Connected graph

A graph is described as **connected** if there is a path between every pair of vertices. In other words, a graph is connected if there is a way to get from any vertex to any other vertex by following its edges. Formally: A graph $G = (V, E)$ is connected iff:

- For all $x, y \in V$ there exists a path $\pi_{(x,y)}$ or $\pi_{(y,x)}$ between them.

If a graph is not connected, it is called **disconnected**. This means that there are vertices from which there is no path to other vertices. As this topic is concerned with warehouse routing, where vertices represent distinct spacial locations in a warehouse, every graph that is referred here can be assumed to be connected. That is, discussed warehouse graphs (or roadmaps) never contain unreachable positions or dead-ends.

2.1.5 Shortest Path Algorithms

Shortest path algorithms aim to find the path between two vertices in a graph such that the sum of traversed edge weights is minimal. Formally, given a weighted graph $G = (V, E, w)$, a paths length is defined as: $\mid \pi \mid = \sum_{e \in \pi} w(e)$.
Let π^* be the set of all possible paths π from vertex x to vertex y. A **shortest path** π' satisfies:

$$\pi' \vdash \forall \pi \in \pi^* : \mid \pi' \mid \leq \mid \pi \mid$$

2.1.6 Optimality (Path Finding)

Optimality refers to the ability of a path finding algorithm to guarantee finding a shortest path π' between two vertices x to vertex y. In the context of MAPF, an optimal algorithm is one that always finds a shortest, collision-free path for all agents.

2.1.7 Completeness (Path Finding)

A pathfinding algorithm is considered **complete** if it guarantees finding a path π if one exists. Such an algorithm only fails if no valid path exists at all.

2.1.8 A-star (A*)

First introduced by (Hart, Nilsson and Raphael, 1968), the **A*** algorithm is a highly efficient pathfinding and graph traversal technique that finds the shortest path from a start vertex to a target vertex with minimal computational expense. It effectively blends Dijkstra's algorithm's thoroughness with the heuristic-driven focus of Best-First-Search.

Algorithm Overview

For each vertex in the graph A* maintains two metrics: the *g*-score, representing the cost from the start to the current vertex, and the *f*-score, representing the estimated cost from the start to the goal through the current vertex. The *f*-score is calculated as: $f(v) = g(v) + h(v)$ where $h(v)$ is a heuristic function that estimates the cost of the cheapest path from v to the goal. It guides the algorithm by estimating which paths are more likely to lead to the goal efficiently.

Admissible Heuristic

A heuristic is **admissible** if it never overestimates the cost of reaching the goal. Together with an admissible heuristic, A* was proven to be optimal by (Dechter and Pearl, 1985). Admissibility ensures that the estimated cost $f(v)$ is a lower bound on the actual cost of the shortest path, which allows A* to iterate over its search-space in a way that it finds shorter paths first.

Euclidean Distance as Heuristic

In scenarios where graphs represent physical space, vertices usually define points in 2D. In such cases, the **euclidean distance**—a straight-line estimate between two points—can be used as an admissible heuristic. For an in-depth explanation, see (Russell, Norvig and Davis, 2010). In the 2D scenario, the euclidean distance is defined as follows:
The distance d between two distinct vertices v_i and v_j in a graph $G = (V, E)$ is defined as: $d(v_i, v_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ where $v_i = (x_i, y_i)$ and $v_j = (x_j, y_j)$.
For a deeper exploration of A* and its applications, the reader is encouraged to consult (Dechter and Pearl, 1985).

2.2 Single Agent Path Finding (SAPF)

Building upon the foundational concepts introduced in the previous chapter on graph theory, this chapter discusses the problem of pathfinding for a single agent in dynamic environments. Unlike in Multi-Agent Path Finding (MAPF), which will be discussed in the next chapter, there is no universally agreed upon terminology for this problem. This problem will be referred to as Single-Agent Path Finding (SAPF).

Traditional pathfinding techniques like A* can find optimal paths very efficiently. However, they are designed for static environments, where the underlying search graph is assumed to be static. Depending on the problem definition this assumption may be sufficient, even when plan execution has to deal with real-world dynamics. One good example for this is the Vehicle Routing Problem (VRP). In the VRP setting, multiple vehicles have to travel an interconnected street network to deliver goods to customers. Even though vehicles may be subject to dynamic delays, the street network is static. In contrast to the MAPF problem, there is no need to prevent multiple vehicles to be at the same location at the same time. Therefore, while searching for every vehicles path, the paths already assigned to other vehicles can be ignored. This key difference allows for the use of traditional pathfinding techniques in the VRP setting.

When such collision avoidance is required, traditional pathfinding techniques are not sufficient anymore and need adaptation. This chapter will discuss the limitations of the traditional pathfinding techniques in dynamic environments and then introduce more advanced techniques like Safe Interval Path Planning (SIPP) that are better suited for this task.

2.2.1 Challenges in Dynamic Environments

In the context of warehouse routing, the underlying planning graph often represents roads in a 2-dimensional space. In such a setting, no two objects or agents can occupy the same location at the same time.

Depending on the problem definition, a warehouse routing system may try to find a path for every single agent independently. Every agent then has to find a path to its goal location without colliding with any dynamic obstacle (e.g. other agents, human workers, or mobile equipment). In this setting, modelling the underlying planning graph as static faces several difficulties:

- **Constant Replanning:** Static pathfinding solutions may require continuous adjustments as new obstacles appear or existing ones move. This means constant replanning is necessary, which can be computationally expensive.
- **Inability to Solve:** If a dynamic obstacle temporarily blocks a path, traditional algorithms might fail to find any viable route. Depending on the problem formulation, this may cause the planning algorithm to return no solution.

2.2.2 Space-Time A*

One traditional solution for planning in dynamic environments is to use Space-Time A*, where time is discretized and treated as an additional dimension in the planning graph. In this approach, the environment is modeled as a 3D space with time as the third dimension. Moving agents plan their paths through this space-time grid, ensuring they do not collide with other agents or obstacles at any given time step.

For further details, the reader is referred to Chapter 2.2.2 (Space-Time A*).

2.2.3 Shortcomings of Space-Time A*

Space-Time A* provides a robust single-agent planner for discrete environments like grid maps used in the classical MAPF formulation discussed in the next chapter (see Chapter 2.3.1 (Classical MAPF)).

In motion planning scenarios, particularly for robotic systems, time often needs to be finely discretized—sometimes down to the frequency of the robot's internal control loop.

In the context of warehouse routing, a planner may find better solutions if space is modeled as continuous. This will be discussed in Chapter 2.3.2.4 (Continous time). Using a discrete planner in such a context requires fine-grained space discretization to approximate good solutions.

In both cases, Space-Time A* faces significant challenges:

- **State Space Explosion:** Fine-grained discretization, whether in time to match a control loop or in space to approximate continuous environments, drastically increases the number of states the planner must evaluate.
- **Memory Intensive:** The expanded state space requires substantial memory to store and manage the vast number of potential states. This can quickly surpass available memory resources, particularly in complex or large-scale environments.

2.2.4 Safe Interval Path Planning (SIPP)

One of the most widely used SAPF techniques is Safe Interval Path Planning (SIPP). The following section explains SIPP in detail. ***Note*** that all explanations are adopted from the original paper by (Phillips and Likhachev, 2011).

As discussed in the previous section, modeling the environment as static is insufficient for finding a path for a single agent in the presence of moving obstacles. While Space-Time A* works well in grid-based environments, it can become computationally impractical when time must be finely discretized or when space is modeled continuously. SIPP avoids the computational pitfalls of adding time as an additional dimension by leveraging safe intervals.

A safe interval is a time interval during which an agent can traverse a specific edge without colliding with any other agent. Every node in the planning graph is associated with a set of

safe intervals. Therefore, in the context of SIPP, finding a path involves identifying a sequence of safe intervals long enough for an agent to travel across them and reach its goal.

2.2.4.1 Definitions

SIPP is a general path planning algorithm that can be adapted to work with different types of agents and environments. To maintain generality, the authors of SIPP deliberately leave agent-specific details undefined, focusing instead on the broader concept of a configuration and its role in path planning.

More formally the **configuration** of an agent, denoted as *cfg*, represents a specific state within that agents configuration space (**C-space**). The C-space is the space of all possible configurations that the agent can assume. A configuration may include the agent's position (x, y), orientation θ, velocity, shape, and other relevant parameters depending on the path planning formulation. For further details on C-space and its relationship to planning algorithms, the reader is referred to (LaValle, 2006).

A **safe interval** is a contiguous period during which a specific configuration *cfg* does not collide with any dynamic obstacles. Conversely, a collision interval is a contiguous period during which the configuration *cfg* is in collision with a dynamic obstacle at each timestep.

An agent's configuration *cfg* has a timeline, which is an ordered list of alternating safe and collision intervals. Therefore, safe intervals are always bounded by collision intervals or infinity.

Since there can never be two consecutive safe intervals, a single **state** s represented by a (configuration cfg^s, interval $I^s[t^s_{\text{start}}, t^s_{\text{end}}]$) pair replaces many states—one for each timestep in the safe interval. This makes the state space significantly smaller, and therefore SIPP more efficient than space-time A*.

2.2.4.2 Notations and Assumptions

For SIPP to work, the following conditions have to be met:

- Trajectories of other agents or dynamic obstacles are known in advance.
- Space is discretized into a Euclidean graph, or in the simplest case, a grid.
- An agent is capable of waiting in place (not true for bipedal vehicles, like motorcycles).
- Kinematic or inertial constraints are negligible. The planner assumes the robot can stop and accelerate instantaneously. (Not necessary for newer versions of SIPP (Ali and Yakovlev, 2023))

Additionally, for SIPP to be optimal, an admissible heuristic function $h(s)$ must exist that estimates the cost from a given state s to the goal state. Since SIPP works in continous time, the cost $c(s, s')$ of an edge-transition from s to one of its successors s' represents the time to execute the action from s to s'. In other words, the goal of the planner is to find a time optimal trajectory.

2.2.4.3 Graph Construction

When the planner is initialized, each graph-node (position in space) is associated with a set of safe intervals. In the beginning each set only contains one single safe interval, from 0 to infinity. This interval represents the availability of nodes across the whole planning horizon. After that, trajectories of known dynamic obstacles are converted to collision intervals and added to their respective graph-nodes. ***Note*** that adding a collision interval is done by splitting the safe interval the collision is contained within into two. Collision intervals are therefore only represented implicitly.

Figure 1 shows an example of adding a collision interval to a graph-node.

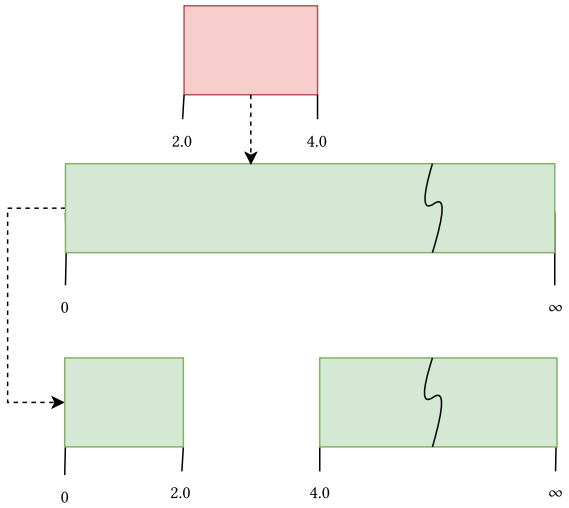


Figure 1: An example of adding a collision interval to a graph-node. Adding the collision (depicted in red) splits the existing safe-interval, leaving two smaller safe-intervals (0.0 to 2.0 and 4.0 to infinity) in its place.

Graph Search

After initialization, an A*-like search (as depicted in Algorithm 1) is used to find the shortest path. The main difference in SIPP is how successor states are generated (line 5) and how time variables of states are updated (line 15). Successor states represent wait and move actions an agent can perform to reach neighboring nodes in the graph from its current location, while respecting its neighbors safe intervals. More specifically, an agent A_G located at a node X with neighbors $N(X)$ may move directly to a neighbor $n(X)$ if the A_G's travel time t between X and $n(X)$ is fully contained within any safe interval of $n(X)$. Agent A_G may also wait in place at node X, if the next safe interval of $n(X)$ fully contains the travel time t and the current safe interval of X is long enough to contain the wait duration until the next safe interval of $n(X)$ starts.

Algorithm 1: Safe Interval Path Planning (SIPP)

input: Graph G, start node s_{start}, goal node s_{goal}
output: Shortest path π: a set of consecutive states $[s_{\text{start}}, \dots, s_{\text{goal}}]$,

$g(s_{\text{start}}) \leftarrow 0$; *OPEN* $\leftarrow \emptyset$; *visited* $\leftarrow \emptyset$
insert s_{start} into *OPEN* with $f(s_{\text{start}}) \leftarrow h(s_{\text{start}})$
while OPEN $\neq \emptyset$:
  remove s with the smallest $f(s)$ from *OPEN*
    if $s = s_{\text{goal}}$
      $\pi \leftarrow s \wedge$ all parents of s (recursive until reaching s_{start})
      return reversed(π)
    successors $\leftarrow$ *getSuccessors()*
    for each s' in *successors*:
      if $\neg s' \in$ *visited*:
        visited $\leftarrow$ *visited* $\wedge\ s'$
        $f(s') \leftrightarrow g(s') \leftrightarrow \infty$
      if $g(s') > g(s) + c(s, s')$
        $g(s') \leftarrow g(s) + c(s, s')$
        updateTime(s')
        $f(s') \leftarrow g(s') + h(s')$
        insert s' into *OPEN* with $f(s')$

Algorithm 2 depicts the generation of successor states in detail:

Algorithm 2: getSuccessors - (SIPP)

input: current state s
output: successors S' : $[s_1, \dots, s_n]$,

$S' \leftarrow \emptyset$
for each a in A(s):
  cfg $\leftarrow$ result of a applied to s
  $t_a^s \leftarrow$ time to execute action(a)
  $t_{\text{start}}^s \leftarrow$ time(s)
  $t_{\text{end}}^s \leftarrow t_{\text{start}}^s + t_a^s$
  for each safe interval i in *cfg*
    if $t_{\text{start}}^i > t_{\text{end}}^s \vee t_{\text{end}}^i < t_{\text{start}}^i$
      continue
    $t \leftarrow$ earliest arrival time of *cfg* during interval i with no collisions
    if t does not exist: continue
    $s' \leftarrow$ state of configuration *cfg* with interval i at time t
    insert s' into S'
return S'

Given an agents current state s all configurations reachable from s are iterated and checked if they fall within any safe interval i associated with that configuration's (*cfg*) current node.

More specifically, for each state s, all possible (move and wait) actions A are considered. In the context of SIPP, an action $a \in A$ maps a state s to a new configuration cfg'. A valid successor state s' is identified if applying a to s results in a configuration and interval pair $\{\text{cfg}', I'\}$ pair that is properly bounded by existing safe intervals i.

Figure 2 illustrates the SIPP successor-state generation process for two agents traveling over a grid map. The (green) agent X is currently at node A and wants to travel to C. The (blue) agent Y moves from B to D. X therefore has to wait one timestep at A because Y has already added a collision interval to B at timestep 0.

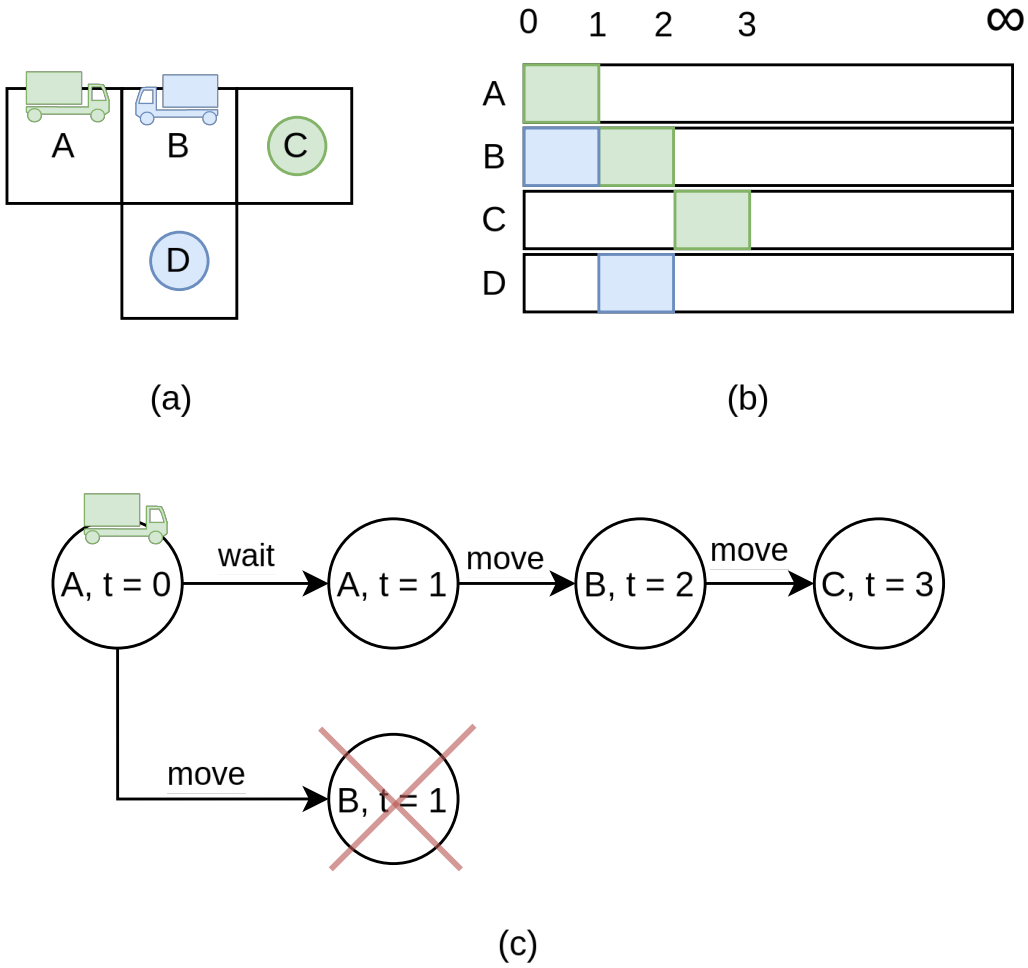


Figure 2: SIPP successor state generation for an agent traveling over a grid map. (a): The environment at t=0, (b): involved collision-interval allocation, and (c): the successor state expansion for green agent X.

Note: While the above mentioned successor-state generation strategy works, depending on the problem definition other strategies may be required. For example, an agent traveling over an edge $(X \rightarrow n(X))$ in continous space may still collide with a dynamic obstacle if its path geometrically crosses over that edge. In such a case, it is not enough to only assume a safe interval in the next node. Scenarios like this will be discussed in Chapter 4 (Design).

2.3 Multi Agent Path Finding (MAPF)

Following a short exploration on single agent path finding, one of the fundamental topics of this thesis is introduced here. This chapter provides an in-depth overview on the basics and theoretical formulations of the Multi-Agent Path Finding (MAPF) problem. Algorithmic strategies and the current state of the art will be discussed in the next chapter.

MAPF involves finding a collision-free path for all agents from their respective starting points to their goal locations. Solving real-time path coordination and collision avoidance, is essential for applications such as robotics and automated warehousing. The substantial interest in MAPF from the research community stems from its practical importance and computational challenges.

(Silver, 2005) first discussed how to solve cooperative pathfinding for multiple agents on a grid map. This work laid the foundation for the MAPF problem. After (Yu and LaValle, 2013) proved that MAPF is NP-hard in general, the problem gained significant attention from the research community.

As research evolved, numerous variations and extensions of the MAPF problem were proposed. Recognizing the need for standardization amidst this growing diversity, (Stern *et al.*, 2019) proposed a unified definition of the classical MAPF problem, and many of its other variants such as MAPF in Euclidean space which is central to this thesis.

2.3.1 Classical MAPF

The following definitions are directly taken from (Stern *et al.*, 2019).

2.3.1.1 MAPF - Definition

The input to a classical MAPF problem with k agents is a tuple $\langle G, s, g \rangle$, where $G = (V, E)$ is an undirected graph, $s : [1, ..., k] \to V$ maps an agent to a source vertex, and $g : [1, ..., k] \to V$ maps an agent to a target vertex. Time is assumed to be discretized, and in every time step each agent is situated in one of the graph vertices and can perform a single action.

An **action** in classical MAPF is a function $a : V \to V$ such that $a(v) = v'$ means that if an agent is at vertex v and performs a then it will be in vertex v' in the next time step. There exist two types of actions: *wait* and *move*. A *wait* action means that the agent stays in its current vertex another time step. A *move* action means that the agent moves from its current vertex v to an adjacent vertex v' in the graph (i.e., $(v, v') \in E$).

For a sequence of actions $\pi = (a_1, ...a_n)$ and an agent i, $\pi_i[x]$ denotes the location of the agent after executing the first x actions in π, starting from the agent's source $s(i)$.

Formally, $\pi_i[x] = a_x(a_{x-1}(...a_1(s(i))))$. A sequence of actions π is a **single-agent plan** for agent i if and only if (iff) executing this sequence of actions in $s(i)$ results in being at $g(i)$, that is, iff $\pi_i[|\pi|] = g(i)$. A **solution** Π is a set of k single-agent plans, one for each agent.

2.3.1.2 MAPF - Conflicts

The goal of a MAPF solver is to find a solution, i.e., a single-agent plan for each agent, that can be executed without collisions. To achieve this, MAPF solvers use the notion of *conflicts* during planning. A MAPF solution is called *valid* iff there is no conflict between any two single-agent plans. The definition of what constitutes a conflict depends on the environment. Common conflict definitions for classical MAPF are listed below. Let π_i and π_j be a pair of single-agent plans.

- **Vertex conflict:**: A *vertex conflict* between π_i and π_j occurs iff according to these plans the agents are planned to occupy the same vertex at the same time step. Formally, there is a vertex conflict between π_i and π_j iff there exists a time step x such that $\pi_i[x] = \pi_j[x]$.
- **Edge conflict:**: An *edge conflict* between π_i and π_j occurs iff according to these plans the agents are planned to traverse the same edge at the same time step in the same direction. Formally, there is an edge conflict between π_i and π_j iff there exists a time step x such that $\pi_i[x] = \pi_j[x]$ and $\pi_i[x+1] = \pi_j[x+1]$.
- **Following Conflict:** A *following* conflict between π_i and π_j occurs iff one agent is planned to occupy a vertex that was occupied by another agent in the previous time step. Formally, there is a following conflict between π_i and π_j iff there exists a time step x such that $\pi_i[x+1] = \pi_j[x]$.

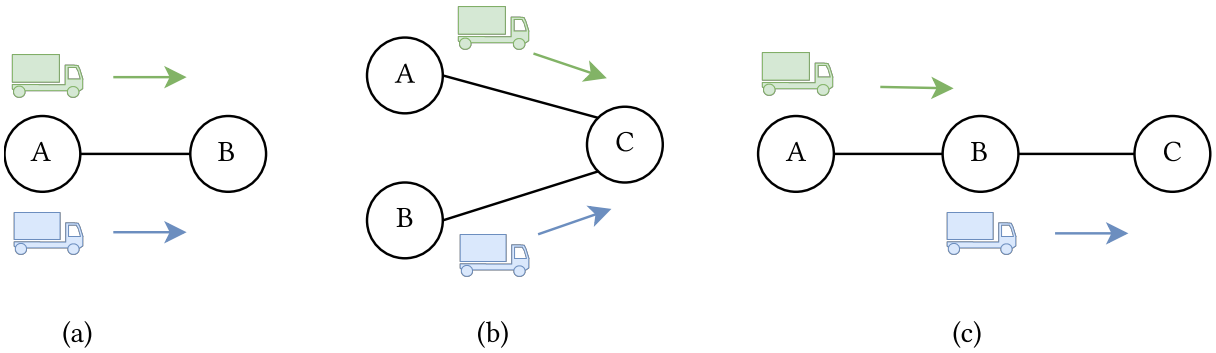


Figure 3: An illustration of common types of conflicts. (a): an edge conflict, (b): a vertex conflict, (c): a following conflict.

Figure 3 illustrates the three above mentioned collision types.

Note that the above mentioned paper also defines a *swap conflict* and a *cycle conflict* which are not considered essential to this thesis. For further details, refer to (Stern *et al.*, 2019).

2.3.1.3 Agent Behavior at Target

In a solution to a classical MAPF problem, agents may reach their targets at different time steps. Therefore, when defining a classical MAPF problem, one must define how an agent behaves in the time steps after it has reached its target and before the last agent has reached its target. There are two common assumptions for how agents behave at their targets.

- **Stay at Target** Under this assumption, an agent waits at its target until all agents have reached their targets. This waiting agent will cause a vertex conflict with any plan that passes through its target after it has reached it. Formally, under the stay-at-target assumption, a pair of single-agent plans π_i and π_j will have a vertex conflict if there exists a time step $t \geq |\pi_i|$ such that $\pi_j[t] = \pi_i[|\pi_i|]$
- **Disappear at Target** Under this assumption, when an agent reaches its target it immediately disappears. This means the plan of that agent will not have any conflict after the time step in which the corresponding agent has reached its target.

2.3.1.4 MAPF - Objective Functions

The two most common functions used for evaluating a solution in classical MAPF are *makespan* and *sum of costs*.

- **Makespan**: The number of time steps required for all agents to reach their target. For a MAPF solution $\Pi = \{\pi_1, ...\pi_k\}$, the makespan of Π is defined as $\max_{\{1 \leq i \leq k\}} |\pi_i|$.
- **Sum of costs**: The sum of time steps required by each agent to reach its target. The sum of costs of Π is defined as $\sum_{1 \leq i \leq k} |\pi_i|$. The sum of costs is also known as *flowtime*.

2.3.1.5 Classical MAPF - Example

Consider the following grid-map scenario illustrated in Figure 4 (a). There are two agents A_B (blue) and A_G (green). Both have their target position t_i at each others starting point s_i. While this is an intuitively simple example, it still requires a collaborative strategy for both agents to reach their target without collision. Figure 4 (b) shows a potential solution to this problem. Each cell represents which agents (A_B or A_G) is at which graph-node $V(A...D)$ at each time step $t(1...6)$.

The *makespan* of this solution (π) is 6, since the last agent A_G reaches its target at $t = 6$.
The *flowtime* is 10, since A_B reaches its target at $t = 4$ and A_G reaches its target at $t = 6$.
In this scenario all three previously discussed conflict types are taken into account. While vertex and edge conflicts are usually included into every MAPF formulation, classical MAPF solvers may or may not consider following conflicts. Here, following conflict constraints prevent agent A_B from moving to vertex v_B, while agent A_G is moving over edge $e_{B \to D}$. If agents would move perfectly synchronous, following constraints would not be necessary. Furthermore, disregarding them would theoretically improve the solution by reducing the makespan to 3. However accurate synchronous movement is not feasible in real-world scenarios, such as AMR routing, which is the focus of this thesis.

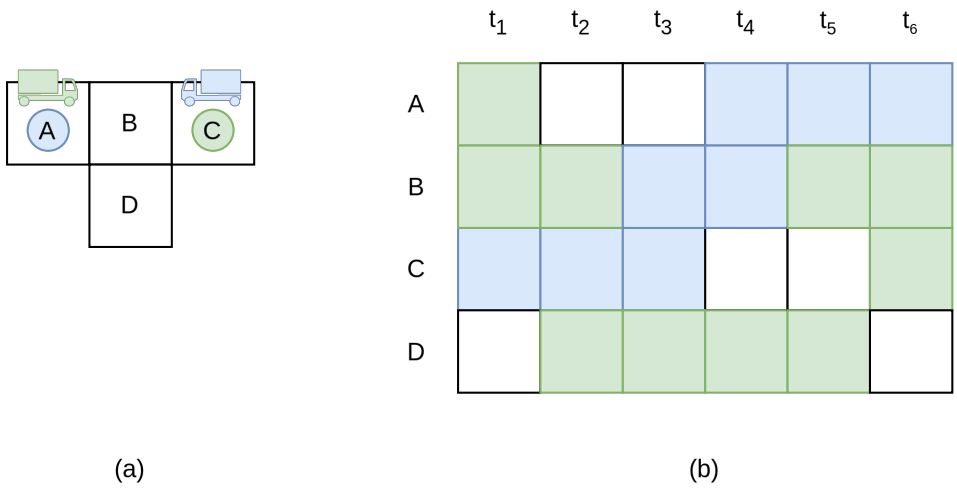


Figure 4: A classical MAPF problem with two agents. (a): the MAPF problem scenario, (b): a potential solution that scenario that takes vertex, edge and following conflicts into account.

2.3.2 Extensions to classical MAPF

The following section was again adopted from (Stern *et al.*, 2019).

In classical MAPF, agents are assumed to occupy exactly one vertex, without shape nor volume and move at constant speed. In contrast, motion planning directly accounts for these physical properties.

At each time step, an agent is situated in a configuration instead of only a vertex. A configuration specifies the agent's location, orientation, velocity, shape etc. (see Chapter 2.2.4.1 (Definitions)). This section discusses a few relevant MAPF extensions, that are a step further to closing the gap between classical MAPF and motion planning.

2.3.2.1 MAPF on Euclidean Graphs.

Continous real-world scenarios such as street networks, or warehouse layouts, are better represented as a directed graph G with vertices V in euclidean space. In this setting, each vertex v_i represents a euclidean point (x,y) in $\mathbb{R}^2$, and edges E represent physical connections such as roads or warehouse hallways between them.

MAPF on Euclidean graphs is a generalization of the classical MAPF formulation. From a euclidean point of view, the classical scenario can be defined as follows:

The underlying graph G consists of equally spaced vertices, forming a 2-dimensional grid, and every edge therefore has uniform length.

As already stated in (Hoenig *et al.*, 2018), since edges and vertices may physically overlap, this setting necessitates additional types of conflicts to consider, namely:

- **Edge-to-Edge conflicts** Two agents may not traverse two edges if a collision could occur during the traversal.
- **Edge-to-Vertex conflicts** One agent may not traverse an edge if a collision could occur with another agent staying at a vertex.

2.3.2.2 MAPF with Large Agents.

In practical applications, agents often possess distinct geometric shapes and volumes. This reality requires adjustments in how an agent's position and movement between vertices are represented within the underlying MAPF graph G. For instance, if an agent is located on one vertex, it may still prohibit other agents from occupying adjacent vertices due to its size. Similarly, if an agent moves along an edge, it may prohibit other agents from moving along intersecting edges or staying at vertices that are too close to the edge.

Large agents can occupy an arbitrary number of entities (nodes or edges) at the same time. Therefore, this setting also requires additional conflicts types, similar to Chapter 2.3.2.1 (MAPF on Euclidean Graphs.).

2.3.2.3 Kinematic constraints

As discussed in (Hoenig *et al.*, 2016), integrating kinematic constraints into the MAPF setting restricts the move actions an agent can perform. There, actions not only depend on an agent's current location, but also on state parameters such as velocity and orientation. In this context agents may not be able to instantaneously change their velocity or orientation and therefore are not always able to move over any edge in any direction. This extension more accurately captures the real-world dynamics of agents. Applying MAPF solutions to real-world AMRs therefore either requires such kinematic constraints to be modeled, or other mechanisms have to be put in place to allow them to follow MAPF solutions. As directly integrating kinematic constraints into the MAPF setting is computationally expensive, this thesis will focus on the latter approach. (Hoenig *et al.*, 2019) describes the conversion of computed single-shot MAPF solution into an action-precedence graph agents can follow, while being able to disregard actual execution times. As this is one of the core algorithmic concepts in this thesis, it will be discussed in more detail in Chapter 4 (Design).

2.3.2.4 Continous time

In the classical MAPF formulation time is discretized and every move an agent can make takes exactly one time step. This simplification allows for a simpler modeling of the problem and therefore more efficient algorithms. However, in the real-world, agents move continuously and may take different amounts of time to traverse different edges. This is especially true for AMRs which may have to slow down or stop before turning or when encountering obsta-

cles. (Varambally, Li and Koenig, 2022) showed that integrating continous time into the MAPF search can lead to more efficient path planning and a higher overall throughput. Depending on the underlying graph structure, modeling the problem with discretized time may not be practical anyway. Consider the following euclidean MAPF example (depicted in Figure 5) where the graph G represents a warehouse layout with differently long hallways. The green agent A_G has to move over a longer hallway (2.5m) than A_B. Even though the agents have different vertices as goals, besides the initial hallways, their paths are symmetrically identical. Independent of the used algorithm, a classical MAPF formulation would not always find a solution that prioritizes A_B over A_G, and thus be suboptimal in solution quality. To address this issue, either non uniform edge costs are necessary, or the problem has to be modeled with continous time.

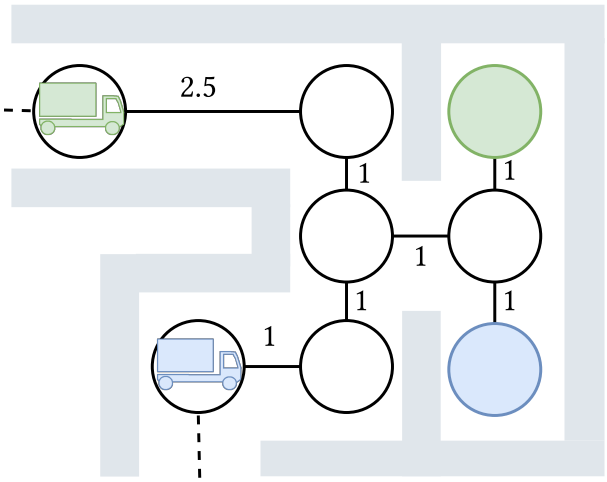


Figure 5: A warehouse layout represented as a euclidean graph MAPF problem, with two agents. The green agent A_G is located on a longer hallway (2.5m) than A_B. A classical MAPF formulation would not always find solutions that prioritize A_B over A_G.

More accurately modeling real-world dynamics can lead to improved solution quality, but also increases the complexity of the problem. Since even the classical MAPF problem is inherently NP-hard, a trade-off between modelling complexity and the used algorithmic strategy is necessary. (Varambally, Li and Koenig, 2022) discusses this trade-off in more detail.

2.4 Algorithmic Strategies for MAPF

Following the previous chapter which introduced the MAPF problem and provided core definitions and extensions, this chapter discusses existing algorithms and solution strategies to tackle MAPF and its many extensions most relevant to this thesis.

2.4.1 Prioritized Planning (PP)

The idea of prioritized planning goes back to (Erdmann and Lozano-Pérez, 1987). In their work, they introduced a simple yet effective approach to coordinate multiple moving objects by assigning priorities to each agent. Planning is done sequentially. Each agent plans its own path while avoiding collision with higher priority agents. This method ensures a collision-free path for all agents, provided that individual paths are feasible.

Specifically, Prioritized Planning (PP) finds a **solution** Π to a given MAPF problem $\langle G, s, g \rangle$ by first assigning a unique priority $[k..1]$ to each of k agents. In descending order each agent i plans its path π_i while avoiding collisions with already existing paths of higher priority agents. Depending on the MAPF definition, these single agent planning steps can be done in several ways. Key considerations are how time is modeled and what shared data structure is used to store already found paths of higher priority agents. In a classical MAPF formulation, agents can use any low-level graph search like Space-Time A*. During search expansion, higher priority agents are treated as dynamic obstacles and their paths π_i are avoided. Since classical MAPF uses also discretized time steps, this simple approach is sufficient.

One of the significant advantages of PP is its scalability. Its planning time scales linearly with the number of agents involved. For further details, the reader is referred to: (Erdmann and Lozano-Pérez, 1987).

2.4.1.1 Optimality and Completeness

Optimality

PP is not guaranteed to produce optimal solutions. The quality of the solution heavily depends on the order of priorities assigned to the agents. Agents with lower priorities may end up with significantly suboptimal paths because they must avoid the paths of all other higher-priority agents. PP is therefore **sub-optimal**.

Completeness

PP is **incomplete**. This means that even if there exists a valid set of collision free paths Π, PP is not guaranteed to find it. In the PP setting, every agent reserves itself the shortest possible path, while avoiding collisions with higher-priority agents. In some cases, this is not sufficient to find a valid solution. Consider the example depicted in Figure 4. There, PP fails as it requires the higher priority agent to move to node D and not reserve the shortest path for itself. Such

collaborative solutions can not be found by PP at all. For an in-depth theoretical analysis the reader is referred to (Ma *et al.*, 2019).

2.4.1.2 Example

Consider the example depicted in Figure 6. Both agents A_G (green) and A_B (blue) want to reach their respective targets. A priority-ordering $P\{A_B, A_G\}$ will fail to find a solution because the higher prioritized agent will not move to the avoidance node F (see Chapter 2.4.1.1 (Optimality and Completeness)). On the other hand, for priority-ordering $P\{A_G, A_B\}$ a valid solution can be found because A_G moves from A to E using the shortest path, while A_B is able to reach the avoidance node at F just in time before A_G moves over D.

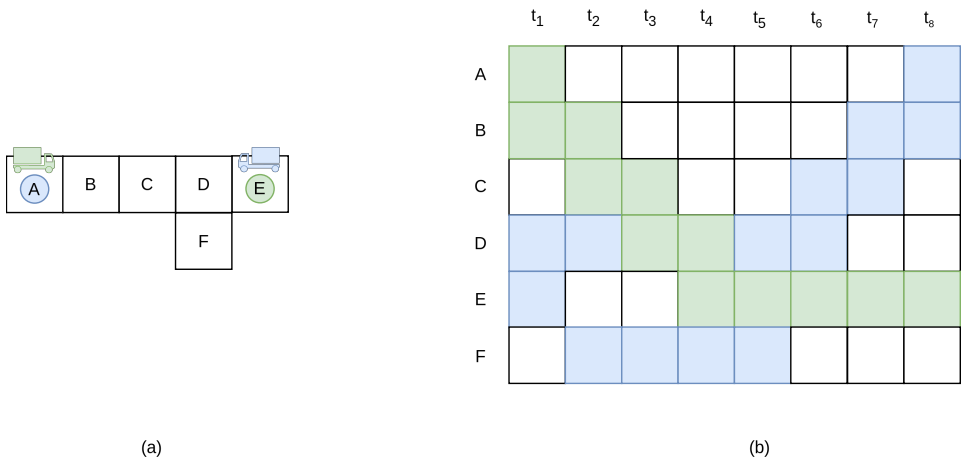


Figure 6: A classical MAPF problem with two agents. (a): The MAPF problem scenario. (The asymmetrical layout allows prioritized planning to find a solution). (b): A potential solution where the green agent has a higher priority than the blue one.

2.4.1.3 Algorithm

A general overview of the basic steps of the PP algorithm is given below:

Algorithm 3: Prioritized Planning (PP)

input: $\langle G, s, g \rangle$, max timeout τ
output: $\Pi : [\pi_1, \dots, \pi_k] \vee \emptyset$
while timeout τ not reached:
  $\Pi \leftarrow \emptyset$
  success $\leftarrow$ true
  prioritizes $P\ [k..1] \leftarrow$ *assignAgentPriorities()*
  for each i in P:
    $\pi_i \leftarrow$ *findPath*$(G, s_i, g_i, \Pi_{\{[1,i-1]\}})$
    if $\pi_i = \emptyset$
      success $\leftarrow$ false
      break

```
10        Π_[i] ← π_i
11    if success
12        return Π
```

As seen in Algorithm 3, the algorithm starts by calling: *assignAgentPriorities()*. This functions provides unique priority orders for each agent. While there are many ways to do this, the most common one is to assign them randomly. Then for each agent, *findPath()* performs a single agent path finding step, taking already found paths $\Pi_{\{[1,i-1]\}}$ into account. The algorithm is repeated indefinitely until *findPath()* succeeds for all k agents or the maximum timeout τ is exceeded.

2.4.2 Extensions to Prioritized Planning

Many extensions to the classical Prioritized Planning (PP) algorithm have been proposed in the literature. The following section discusses some of the most important ones relevant to this thesis.

2.4.2.1 Deterministic Rescheduling

Many different strategies can be used to generate priority ordering. In (Andreychuk and Yakovlev, 2018) the authors discuss the use of a deterministic rescheduling algorithm. There, after an initial ordering failed, new priority orderings are generated by setting the failed agent to the highest priority and shifting all others to the right. Therefore, in the next iteration, the previously failed agent will not fail to find a path anymore. For a higher number of agents this approach needs significantly less iterations to find a successful priority order than just random assignment.

2.4.2.2 Safe-Start Intervals (SSI)

Also introduced in (Andreychuk and Yakovlev, 2018), Safe Start Intervals (SSI) work on the insight that a lot of planning failures are due to the fact that higher-priority agents directly move over the start locations of lower-priority ones without enough time for them to move out of the way.

To circumvent this problem, SSI therefore prohibits the movement of any agent over the start location of another agent for a certain amount of time. Consider the example depicted in Figure 7: Both agents A_G (green) and A_B (blue) are in the way of each other. Using prioritized planning, two priority orderings are possible: $\Pi = \{A_G, A_B\}, \{A_B, A_G\}$. If agent A_G goes first, A_B has no chance to move out of the way and planning fails. Because this example considers following conflicts, if agent A_B goes first, A_G also can't move out of the way fast enough. Using SSI with k = 1, both agents have the first interval reserved for themselves. This way, if A_B goes first, it will wait at its start position for one time step. This gives A_G enough time to move out of the way and the planning will succeed.

For further clarification, Figure 7 results in following shuttle plans:

$\pi_G = \{\text{move}(B, D), \text{wait}(D)\ \text{wait}(D), \text{move}(B, C), \text{stay at}(C)\}$

$\pi_B = \{\text{wait}(C), \text{move}(C, B), \text{move}(B, A), \text{stay at}(A)\}$

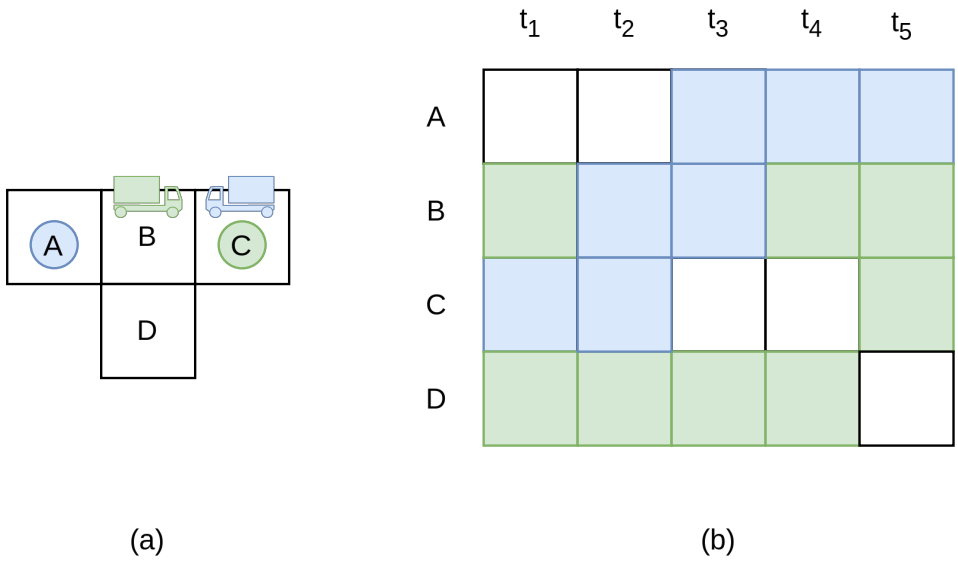


Figure 7: A prioritized planning scenario, solvable with Safe Start Intervals (SSI). (a): The MAPF problem scenario. (b): A potential solution found by Safe-Start Interval k = 1.

2.4.3 Prioritized Safe Interval Path Planning (PSIPP)

While the classical MAPF formulation works on grid-maps and discretized time intervals, some researchers have explored the potential for better solutions using a continous time formulation. (Kasaura, Nishimura and Yonetani, 2022) introduced the Prioritized Safe Interval Path Planning (PSIPP) algorithm. In essence PSIPP is an extension of the classical PP algorithm that aims to solve euclidean MAPF problems. (See: Chapter 2.3.2.1 (MAPF on Euclidean Graphs.)). It does this by first pre-computing Continuous Time Conflict (CTC) and then using those together with SIPP (as the low-level single agent planner) for an efficient continous time collision avoidance strategy.

2.4.3.1 PSIPP Algorithm

The PSIPP algorithm works generally as follows: ***Note*** that for a complete description the reader is referred to (Kasaura, Nishimura and Yonetani, 2022).

Pre-Computing Continuous Time Conflicts (CTC)

Given an existing euclidean roadmap graph G_R, every entity in the graph (nodes and edges) is iterated and compared to all of its neighbors, within a predefined radius.

A Continuous Time Conflict (CTC) is then found if an agent staying at a node or moving over an edge will geometrically collide with another agent, staying or moving on one of its neighbors. Their key insight is that these conflicts can be precomputed only using geomet-

ric algorithms. This prevents the underlying SIPP planner from having to check intervals for every entity in the graph at every planning step. If CTCs are found, respective graph entities are then annotated with them.

Solving the MAPF Problem

Once all CTCs are precomputed, solving the MAPF problem is done like in Algorithm 3. The only difference is how ***findPath()*** works.

A path π_i for an agent i is found by using SIPP (See: Algorithm 1). Standard SIPP computes successors by checking if adjacent intervals are free for neighboring nodes in the grid. In the PSIPP setting however, previously found CTCs are used to determine which neighboring entity (node or edge) needs to be looked at for successor state generation.

2.4.3.2 PSIPP Example

Suppose there is the following roadmap graph G_R with agents A_G and A_B depicted in Figure 8. First all graph entities (vertices V and edges E) get annotated with CTCs. This results in node A having conflicts with its adjacent edges $(A - C)$ and $(A - B)$. Because if an agent stays at A it would collide with another agent moving over any of these edges. Similarly, edge $(A - C)$ has the following conflicts: $\{A, C, (B - D)\}$. A and C are obvious, but a conflict with $(B - D)$ can only be found with geometric collision checking.

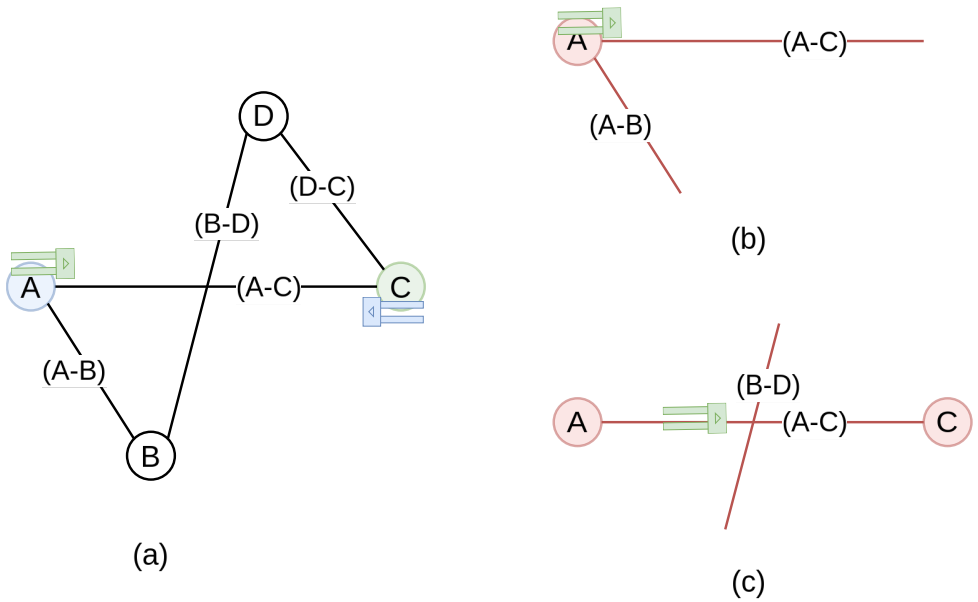


Figure 8: An euclidean MAPF problem with two agents. Both agents have to move over the roadmap to their respective goal nodes. Agents can not move over (B-D) and (A-C) at the same time because they geometrically overlap. (a): The euclidean MAPF problem scenario. (b): An agent staying at node A blocks all adjacent edges. (c): An agent traveling over edge (A-C) block all overlapping nodes and edges.

After pre-computing all CTCs planning problems can be solved efficiently. When SIPP then tries to find a path from A to C safe intervals are only checked for any entity the agents current position shares a CTC with. In this way euclidean graph neighborhood search becomes computationally much more feasible.

2.4.4 Conflict-based Search (CBS)

The CBS algorithm is a widely used algorithm for solving multi-agent pathfinding (MAPF) problems. It was first introduced by (Sharon *et al.*, 2012). CBS aims to combine the optimality of a global A* search with the efficiency of single agent search algorithms. To achieve this, CBS leverages a two-level algorithmic approach. On the higher level CBS builds a Constraint Tree (CT) representing conflicts between agents. On the lower level, CBS uses as single-agent search algorithm, where each agent plans its path independently, while still considering, constraints from the higher level.

One of the reasons for the popularity of CBS is its provable optimality and completeness. However, like every optimal MAPF solver, CBS also has an exponential runtime in the worst case. The main advantage over a global A* search is that CBS is exponential in the number of conflicts, not in the number of agents. This makes CBS better scalable to a larger numbers of agents. The state-of-the-art CBS version is Explicit Estimation CBS (EECBS). It relaxes the optimality of CBS by a user-defined bounded factor. This allows EECBS to leverage focal search. Together with multiple symmetry-breaking techniques (Li *et al.*, 2020) and online learning (to estimate inadmissibility), EECBS scales up to hundreds of agents. For further details, the reader is referred to: (Li, Ruml and Koenig, 2021).

2.4.4.1.1 CBS Algorithm

As described in (Sharon *et al.*, 2012), CBS considers a classical MAPF problem where the graph represents an evenly spaced grid and time is discretized into time steps. Following definitions are directly taken from there. For the sake of comparison, their notation style is adopted directly. Clarifications are provided where necessary.

Algorithm 4: Conflict Based Search (CBS)

input: $\langle G, s, g \rangle$, max timeout τ
output: $\Pi : [\pi_1, \dots, \pi_k] \vee \emptyset$

```
R.constraints ← ∅
R.solution ← find individual paths using low-level()
R.cost = SIC(R.solution)
OPEN ← {R}
while OPEN not empty:
    P ← best node from OPEN
    Validate the paths in P until a conflict occurs
    if P has no conflicts:
```

```
9       return P.solution
10    C ← first conflict (a_i, a_j, v, t) in P
11    foreach agent a_i in C:
12      A ← new node
13      A.constraints ← P.constraints + (a_i, s, t)
14      A.solution ← P.solution
15      Update A.solution by invoking low-level(a_i)
16      A.cost ← SIC(A.solution)
17      OPEN ← OPEN + A
```

At the high-level, CBS searches a CT. A CT is a binary tree. Each node N in the CT contains the following data:

- **A set of constraints** (N.constraints)

  The root of the CT contains an empty set of constraints. The child of a node in the CT inherits the constraints of the parent and adds one new constraint for one agent.

- **A solution** (N.solution).

  A set of k paths, one path for each agent. In Algorithm 4 this is depicted as **low-level()**.

  Note that a *solution* is only equivalent to Π (see Chapter 2.3.1.1 (MAPF - Definition)) if it does not contain any conflicts.

- **The total cost** (N.cost).

  (summation over all the single-agent path costs.) This metric is used as the f-value of the node. In Algorithm 4 this is depicted as **SIC()** (summation of individual costs).

Processing a node in the CT: Given the list of constraints for a node N in the CT, the low-level search is invoked. This search returns one shortest path for each agent, a_i, that is consistent with all the constraints associated with a_i in node N. Once a consistent path has been found for each agent with respect to its constraints, these paths are then validated with respect to the other agents. ***Note*** that since the standard CBS formulation uses discretized time steps, this validation is trivial and only involves checking if no two agents are at the same vertex at the same time step. If all agents reach their goal without any conflict, this CT node N is declared as the overall best solution Π to this MAPF problem $\langle G, s, g \rangle$ and returned by CBS. If, however while performing the validation a conflict $C = (a_i, a_j, v, t)$ is found for two or more agents a_i and a_j, the validation halts and the node is declared as a non-goal node.

Resolving a conflict: Given a non-goal CT node N whose solution N.solution includes a conflict $C_n = (a_i, a_j, v, t)$, in any valid solution at most one of the conflicting agents (a_i and a_j) may occupy vertex v at time t. To resolve a conflict either agent a_i or a_j must be constrained from occupying vertex v at time t. This is done by either adding constraint (a_i, v, t) or constraint (a_j, v, t) to the set of constraints in N.constraints.

To guarantee optimality, both possibilities are examined and N is split into two children A and B. Both A and B inherit all constraints from N. A resolves the found conflict by adding the constraint (a_i, v, t) and B by adding (a_j, v, t). Note that for a given CT node N, one does not have to save all its cumulative constraints. Instead, it can save only its latest constraints and extract the other constraints by traversing the path from N to the root via its ancestors. Similarly, with the exception of the root node, the low-level search should only be performed for agent a_i which is associated with the newly added constraint. The paths of other agents remain the same as no new constraint was added for them.

2.4.4.2 CBS Example

Consider the example depicted in Figure 9, below. Two agents A_G (green) and A_B (blue) want to move to their respective goal locations. More formally the MAPF situation depicted on the top left corresponds to the following plan: A_G - $[D2 \rightarrow A3]$, A_B - $[C1 \rightarrow B4]$. Initially no single-agent paths exist. The **optimal path routes** on the top right depict all possible single agent paths that can initially be found by the low-level search. Which one of them the low-level search returns is up to chance as the f-cost for all of them is the same (8). The underlined single agent-paths are the ones this example is based one. This initial solution is added as the root node of the constraint tree. Then during solution validation, the algorithm steps through each agents individual path simultaneously and finds a collision at $t = 4$ (timestep 4) where both agents A_G and A_B plan to be at node B3. Since the initially found solution is not valid, the algorithm splits the root node into two children. A left child node (indicated in Figure 9 with a 2) is added with constraint (A_G, B3, 4) (disallowing agent A_G to traverse B3 at t=4). Similarly a right child node (3) is added, disallowing A_B from traversing B3 at t=4. The algorithm repeats by invoking the low-level search for Constraint Tree (CT) nodes, based on the f-cost of the nodes. After all nodes with an f-cost of 8 have been expanded, the algorithm finds a solution without conflicts (e.g. node (4)) and terminates.

Note that which node with the same f-cost is iterated first depends on when it is added to the OPEN set and is thus given by chance. Figure 9 depicts the full Constraint Tree (CT) expansion for clarification purposes. For further details, the reader is referred to the educational video from which this example was directly taken: (Symons, 2017)

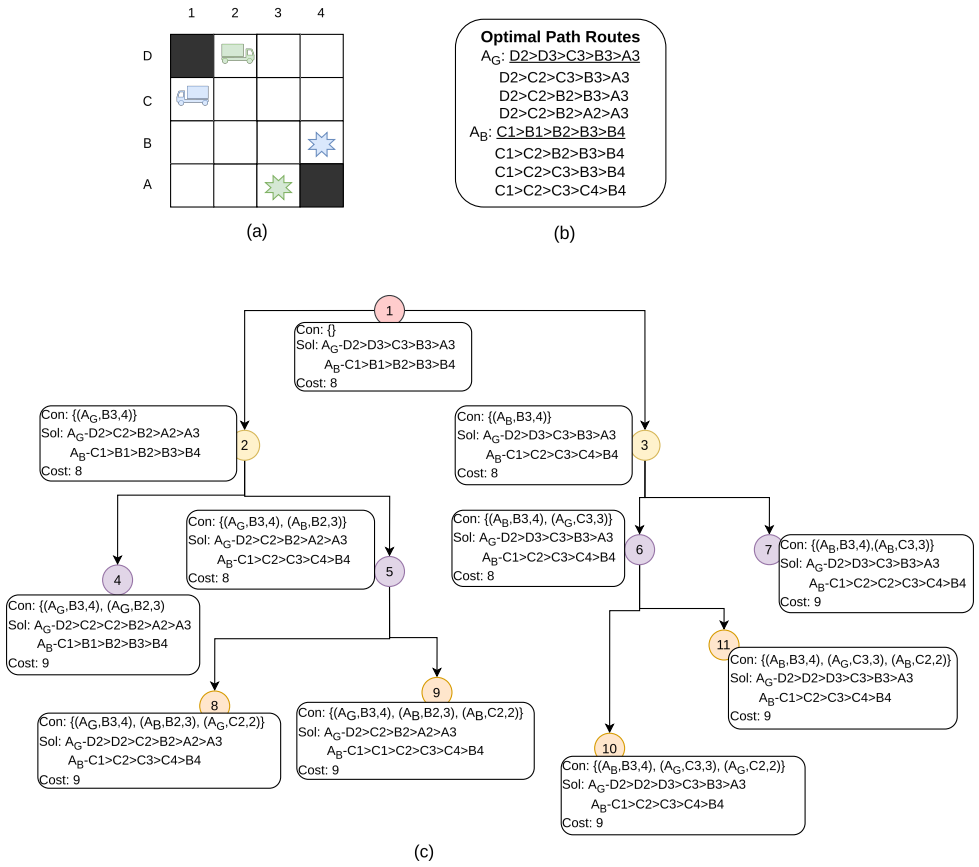


Figure 9: MAPF problem with two agents A_G (green) and A_B (blue) highlighting how CBS finds the optimal solution. (a): The MAPF situation with both agents and their respective goal nodes (indicated with a star). (b): All optimal paths that could be found by the initial low-level() search. (The picked ones are underlined). (c): The CBS - Constraint tree expansion.

2.4.5 Continuous-time Conflict-based Search (CCBS)

The original CBS mentioned uses the classical MAPF formulation Chapter 2.3.1 (Classical MAPF) and thus assumes that time is discrete, with any agent's motion between vertices always taking one timestep. In this model, the graph G follows a grid-map/lattice-like pattern. This is a common assumption in MAPF literature as it simplifies the problem and allows for efficient algorithms. However, in real-world scenarios, such simplifications may lead to suboptimal solutions (e.g., see section: Chapter 2.3.2.4 (Continous time)).

Continuous-Time Conflict-Based Search (CCBS) aims to solve a more general MAPF problem. In this setting, the planning graph G is euclidean (see Chapter 2.3.2.1 (MAPF on Euclidean Graphs.)), meaning the time it takes to move between to vertices v_i and v_j depends on the distance between them. To find an optimal solution CCBS considers time as continous. To achieve this, the authors (Andreychuk *et al.*, 2022) incorporated SIPP (see Chapter 2.2.4 (Safe Interval Path Planning (SIPP))), as a continous time low-level planner, into the two level Constraint Tree (CT) framework of CBS. For further details, the reader is referred to the original paper by (Andreychuk *et al.*, 2022), where this chapter is based on.

2.4.5.1 Conflicts in CCBS

In CCBS, a conflict is not only defined by a vertex, timestep pair (v,t), but can happen between any two agents at any time traveling over any edge or staying at a vertex. Thus, the authors defined conflicts a CCBS as conflicts between actions (see Chapter 2.3.1.1 (MAPF - Definition)). More formally, a conflict is a tuple $\langle a_i, t_i, a_j, t_j \rangle$ representing that if agent i executes action a_i at time t_i and agent j executes action a_j at time t_j, then they will collide. Collision detection between single agent plans is done by checking whether plan execution leads to physical overlap of agents' trajectories. As this process is not straightforward and depends not only on the agents shape but also on its kinematic constraints, the authors use a simplified model considering agents to move with constant speed and all have the same shape. In this setting, collision detections can be done geometrically. If a collision is found a conflict is added to the CT.

2.4.5.2 SIPP for planning with CCBS constraints

To adapt SIPP to be used as a low-level solver for CCBS, actions violating constraints are prohibited during SIPPs search expansion. More specifically, let $\langle i, a_i, [t_i, t_i^u] \rangle$ be a constraint imposed over agent i, to plan a path for agent i with SIPP two cases are distinguished:

Found **action a_i is a move action**: Let v and v' be the source and target destinations of a_i. If the agent arrives to v in time step $t \in [t_i, t_i^u]$ then the action a_i is removed and an action representing waiting at v until t_i^u is added.

Action a_i is a wait action: Let v be the vertex in which the agent is waiting in a_i. Waiting at v is prevented by adding a collision interval to node v for interval $[t_i, t_i^u]$.

2.4.5.3 CCBS Example

To further clarify how CCBS works consider example in Figure 10 adopted from (Andreychuk *et al.*, 2022). Two agents A_G (green) and A_B (blue) travel over a shared euclidean roadmap. Initially CCBS computes paths for each agent using SIPP. Since agents are assumed to move at a constant speed of 1 $\frac{m}{s}$ with instantaneous acceleration, the time it takes for an agent to move over an edge is equal to the euclidean distance between the edge's endpoints. SIPP, therefore, finds paths with time intervals corresponding to the nodes distances. Figure 10 depicts this initial paths in the top right square. Since both agents start moving over overlapping edges (F-I) and (H-C) at the same time t=2, the initially conflict free solution would lead to a collision.

Note: the original paper considers a geometry collision detection mechanism to determine unsafe intervals for both agents. As already discussed, in real-world scenarios this depends on kinematic motion models and the instantaneous acceleration assumption may provide conflict free paths that are not feasible in reality. A simpler collision detection mechanism used throughout this thesis is to block other agents from moving over intersecting edges until the agent has fully finished its move action. Of course this will lead to suboptimal solutions, as the

edge in question can be arbitrarily long. In the context of this example this means that either, A_G will be blocked from moving over edge (F-I) until t=7.0, or A_B will be blocked from moving over edge (H-C) until t=4.828. On the bottom Figure 10 depicts the CT expansion of the CCBS algorithm. Adding a conflict for above mentioned unsafe intervals, results in two child nodes. Both of these child nodes are valid solutions as they themselves do not have anymore conflicts. The algorithm picks the right child as its solution as it has the lowest sum of costs.

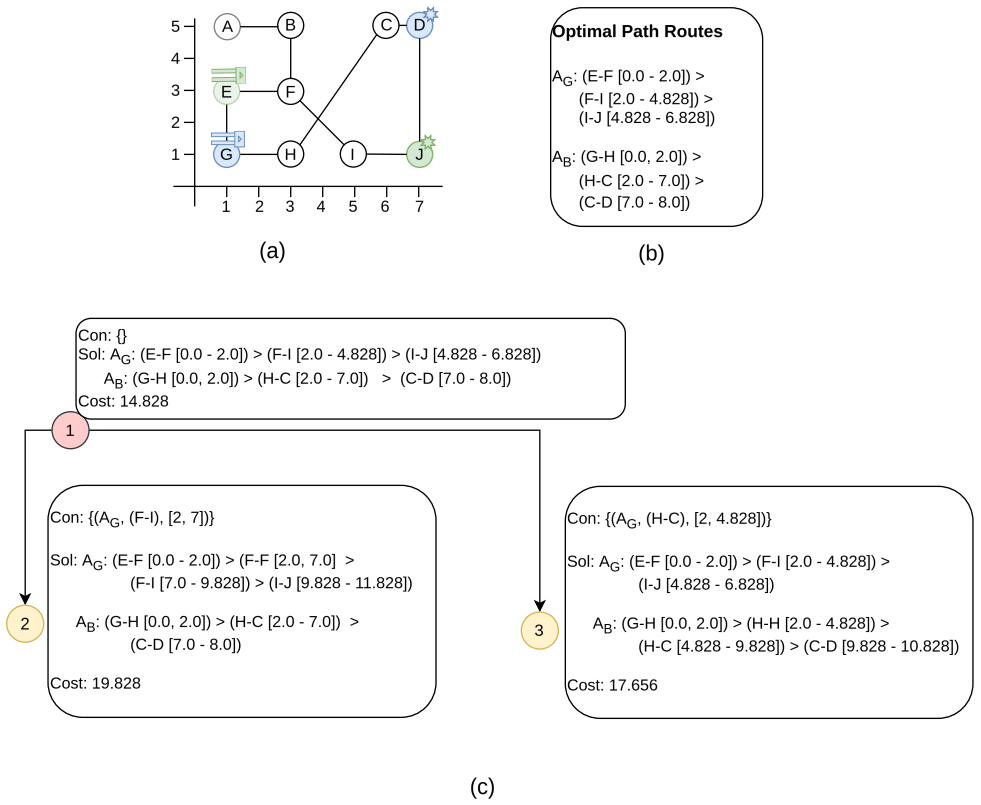


Figure 10: An euclidean MAPF problem and its corresponding CCBS solution. (a): the MAPF problem with two agents A_G (green) and A_B (blue), (b): Optimal paths that could be found by the initial low-level() search. (c): The CCBS - Constraint tree expansion.

2.4.5.4 Disadvantages of CCBS

While CCBS is a powerful algorithm that generalizes the classical MAPF formulation to a more realistic setting, it does not scale as well. This is the case because in the continous setting many more conflicts can occur. Since CBS-based methods scale exponentially with the number of conflicts, this fatally interferes with CCBS's performance. Even though the authors suggested multiple strong improvements in (Andreychuk *et al.*, 2021), depending on the scenario, CCBS may still not be able to find a solution in a reasonable amount of time for more then 30 - 50 agents.

2.5 Multi-Agent Pickup and Delivery (MAPD)

In Chapter 2.3 (Multi Agent Path Finding (MAPF)) the MAPF problem was discussed. Although solving it is a significant challenge, it does not fully capture many aspects of real-world domains, such as task allocation and continuous operation in automated warehouses. The Multi-Agent Pickup and Delivery (MAPD) problem therefore extends MAPF by assigning pick-up and delivery tasks to agents in an online, lifelong manner. With many sophisticated MAPF algorithms now existing, academic interest has recently started shifting towards the MAPD problem. This chapter will provide a brief theoretical background on the MAPD problem and define all terms relevant to this thesis. For further information on the topic, readers are referred to (Ma *et al.*, 2017). Note that the following sections will directly use definitions and terms from this paper.

2.5.1 Characteristics of MAPD

As mentioned above MAPD incorporates several additional aspects of real-world operations. For instance, in robotic warehouse operation, agents receive a **life-long** (never ending) stream of pick-up and delivery tasks (tasks may depend on each other). These tasks usually arrive **online** (whenever a new item needs to be fetched) at **any time** during execution. The following section describes these definitions in detail.

2.5.1.1 Lifelong Operation

Lifelong refers to the continuous, ongoing nature of the problem. A MAPF problem is considered solved once all agents reach their destinations, Unlike the one-shot formulation MAPD operates indefinitely. This necessitates balancing short-term efficiency with long-term performance. An optimal solution to a MAPD problem must take all available tasks into account. Every shuttle may receive an arbitrary number of tasks that may depend on each other. Therefore the solution quality not only depends on each agents path but also on agent-task assignment.

2.5.1.2 Online Task Assignment

Online refers to the fact that tasks are not known in advance but appear dynamically during operation. This is in contrast to an **offline** problem where all information is available at the start. In a real-world setting like a delivery service, new orders (tasks) come in continuously while agents are already executing previous tasks. This online characteristic introduces a crucial challenge:
Algorithms must make decisions with incomplete information about future tasks. This can lead to high-quality solutions being suboptimal in the long-term.

2.5.1.3 Anytime Task Arrival

Anytime describes the fact that tasks can arrive at any time during operation. This further complicates agent operation and can cause different edge-cases related to agent-task assignment. For example, when warehouse operation is started only a limited amount of tasks may be available for assignment and not all agents may receive one. Assigned tasks then may have an idle agents start position as its goal. Executing such tasks will not succeed until the idle shuttle will itself receive a task or other measures like sending it away or doing a task swap are taken. In fact similar situations can happen whenever there are no tasks available for assignment.

2.5.2 Problem Definition

Similar to Chapter 2.3.1 (Classical MAPF), a MAPD problem consists of m agents $A = \{a_1, a_2, ..., a_m\}$ traveling over a graph $G = (V, E)$ where vertices V correspond to locations and edges E to connections between locations that agents move along. At each timestep t_i any number of tasks may arrive. Let T be the set of all tasks that will arrive over the problems lifespan. Each task $\tau_j \in \mathrm{T}$ is then characterized by a pickup location $s_j \in V$ and a delivery location $g_j \in V$. An agent is called free (or idle) if it is currently not executing any task. Otherwise, it is called occupied. Any free agent a_n can be assigned to any task τ_j. It then has to move from its pickup location s_j to the task's delivery location g_j. When the pickup location is reached the agent performs the task, τ is removed from T and the shuttle is free again.
Tasks may be swapped or re-assigned between any agent as long as that agent has not been at the pickup-location already.

2.5.3 MAPD objective

(Ma *et al.*, 2017) considers an average number of time steps per task (or *service time*) as the primary objective. If T is finite (e.g. in simulation) *makespan* and *flow-time* may still be used as performance metrics.

2.6 Bridging the Gap between Theory and Practice

In this chapter first real-world warehouse system requirements are discussed. Then the Rolling Horizon Collision-Resolution and Execution (RHCRE) framework and its components (Action Dependency Graph (ADG) and Rolling Horizon Collision Resolution (RHCR)) are discussed in detail. As the RHCRE framework is central to this thesis, a special emphasis is put on how it can bridge the gap between theoretical MAPD research and real-world applications.

2.6.1 Real-world application needs

As discussed in chapter Chapter 2.5 (Multi-Agent Pickup and Delivery (MAPD)), the MAPD problem generalizes the classical MAPF problem to capture many aspects of real life ware-

house logistic systems. To be more specific: It incorporates the interdependence between pick up and delivery tasks, the fact that tasks may arrive at any time (online), and the continous ongoing (lifelong) nature of warehouse operation.

Besides this, real-world warehouse AMR systems, unfortunately face some additional challenges not directly covered by the MAPD problem formulation. These challenges are:

- **Fault tolerance**: Sometimes AMR fail to execute a task, or even fail to move at all. In such cases, a warehouse routing system should, for instance, be able to navigate traffic around the blocked area until the failed AMR is operational again.
- **Robustness to operation delays** Previously mentioned planners either used a simplified discrete time formulation (CBS, PP) or a continuous time formulation (PSIPP, CCBS), but all of them assume agents to be able to follow plans according to the estimated time. In reality, AMR system can never have such guarantees and slight delays happen all the time. A warehouse routing system, therefore has to allow for time independent plan execution.

2.6.2 Robust execution of plans

To address the challenge of executing a MAPF plan on real life AMRs with real life uncertainty, three main ideas emerged in reasoned years:

- **Integrate operational uncertainty into the planning step**: This includes methods like those in: (Zhang *et al.*, 2024) or (Atzmon *et al.*, 2018), where the used MAPF planner finds k-robust plans Π_k where each agent's operational delay will not cause conflicts for k time steps.

While an elegant solution, finding a k-robust is more computationally expensive.

- **Predictively control agents during execution**: In this category fall works such as: (Gregoire, Čáp and Frazzoli, 2017). Here, agents are continuously sent predictive velocity commands to ensure they stay in their determined time-intervals. If this is not possible, several agents can be slowed done or in the worst case planning re-executed. While this approach works it needs persistent communication between agents and the planner, which is not always feasible and does not scale well to a large number of agents.
- **Post-process a plan into a time independent formulation**: Several such methods have been proposed, most notably MAPF-POST (Hoenig *et al.*, 2016) and the Action Dependency Graph (ADG) framework (Hoenig *et al.*, 2019). Both methods leverage the idea that a MAPF plan implicitly encodes dependencies between agents. After a MAPF plan is found, MAPF-POST converts this plan into a temporal-network. Agents use this network to follow their plan and monitor their progress. If a delay occurs, agents communicate necessary network changes with each other. The ADG framework mitigates the need for continous communication between agents and the planner by converting a MAPF plan into an action-precedence graph. In this setting agents only have to process their own queue of actions and notify the

system if an action is completed. This simple and elegant approach works so well that it was adopted by (Varambally, Li and Koenig, 2022) for their RHCRE framework.

2.6.3 The Action Dependency Graph (ADG) framework

The Action Dependency Graph (ADG) framework is a robust execution method designed to address the challenges of executing MAPF plans in real-world scenarios with operational uncertainty. As already discussed real-world agents (or robots) are not always able to exactly follow their planned time-intervals. In the ADG framework a found MAPF plan is converted into an action precedence network (or action dependency graph), that agents then can follow independently of how long individual actions actually take. This not only minimizes the need for continous communication of the agents and their current state but also allows robust plan execution in the presence of operational uncertainty. Another advantage of the ADG framework is that by extending an existing ADG with a new plan, the ADG framework allows for overlapping of planning and execution, effectively reducing agent downtime in the face of long centralized planner computation times such as CBS.

2.6.4 Construction of the ADG

For a given MAPF solution $\Pi = (\pi_1, ...\pi_k)$, the ADG framework constructs an action precedence graph. The ADG construction algorithm consists of two steps:

1. **Vertex and Type 1 Edge Creation**: Each action $A(a_1^i \ ... \ a_k^i)$ in every agent plan π^i is represented as an action-vertex p_k^i in the ADG. Each vertex p_k^i contains the following information:
   - The action a_k^i itself. (action type: {move, wait}, start vertex s_k^i, goal vertex g_k^i)
   - The time step t_k^i at which the action is executed.
   - The vertices state q_k {*completed*, *enqueued*, *pending*} used for plan execution and communication.

After all action-vertices are created with q_k set to *pending*, type 1 edges (inner-agent dependencies) are created between all vertices p_k^i of the same agent i. These edges represent the order in which actions should be executed. More specific: For a plan π^i, for each action a_k^i an edge $a_k^i \rightarrow a_{\{k+1\}}^i$ is added.

2. **Type 2 Edge Creation**: In the second step, type 2 edges (cross-agent dependencies) are created by exhaustively iterating over all vertices and adding an edge between vertices $(p_k^i \rightarrow p_{k'}^{i'})$ for different agents (i, i') if the start node s_k^i of p_k^i is $p_{k'}^{i'}$'s goal node $g_{k'}^{i'}$ and $t_k^i \leq t_{k'}^{i'}$.

2.6.4.1 ADG Construction Algorithm

Algorithm 5 depicts the whole ADG construction algorithm, adopted from (Hoenig *et al.*, 2019).

Algorithm 5: Action Dependency Graph (ADG) Construction

input: Plan π^i for each robot (or agent)
output: G_{ADG} (V_{ADG}, E_{ADG}) - Action Dependency Graph
/* *create vertices and Type 1 edges* */
foreach agent i:
  Add vertex p_1^i to V_{ADG}
  $p \leftarrow p_1^i$
  for $k \leftarrow 2$ to n_i:
    Add vertex p_k^i to V_{ADG}
    Add edge $p \rightarrow p_k^i$ to E_{ADG}
    $p \leftarrow p_k^i$
/* *create Type 2 edges* */
foreach agent i:
  for $k \leftarrow 1$ to n^i:
    foreach agent i':
      if $i \neq i'$:
        for $k' \leftarrow 1$ to $n^{i'}$:
          if $s_k^i = g_{k'}^{i'}$ and $t_k^i \leq t_{k'}^{i'}$:
            Add edge $p_k^i \rightarrow p_{k'}^{i'}$ to E_{ADG}
            break

2.6.4.2 Plan execution

After constructing G_{ADG}, the ADG framework allows for agents to easily execute their plans, without much communication. Conceptually it works as follows:

Each agent i has its own action queue Q^i that contains all action-vertices p_k^i that are in state *enqueued*. Initially and whenever an agent i finishes an action a_k^i and sets the state of p_k^i to *completed*, the ADG framework recursively checks all direct dependencies of p_k^i and sets them to state *enqueued* if possible. ***Note*** that an action vertex p_x^y can only be set from *pending* to *enqueued* iff all other vertices that have an outgoing edge to it are themselves in state *completed* or *pending*. Throughout this thesis this subroutine will be called: **enqueue_vertices()** .

2.6.5 Lifelong planning with the ADG framework

In the context of lifelong planning, using the ADG framework has additional advantages beyond robust plan execution. In their paper (Hoenig *et al.*, 2019), the authors describe how to overlap planning and execution of multiple plans by extending an existing ADG with a new plan. This allows for agents to continue executing their current plan even while a new plan is being computed. This effectively reduced agents downtime that would be caused by a slow centralized planner such as CBS. This works as follows:

After an initial plan is created agents may execute their individual actions. If one agent i signals that it has only m (user defined) actions left, a replanning can be triggered. The idea is

then to find a ***commit cut*** c_i for the current G_{ADG} that is as far in the future as possible without breaking any dependencies. The ***commit cut*** c_i is then used to partition G_{ADG}. This means that any action after c_i is removed from G_{ADG}. Then, the last actions for every agent i are used as start nodes to solve the next MAPF problem. The new solution Π' is then converted into a new $G_{\text{ADG}}{}'$ and concatenated with the old existing G_{ADG}. Meanwhile agents can continue executing all their actions left in the old G_{ADG}. This way planning can effectively be done ahead of time before agents are actually finished with their current paths. This process can be repeated indefinitely, effectively overlapping planning and execution.

2.6.5.1 Commit Cut Computation

This chapter shortly describes how the commit cut c_i for a given graph G_{ADG} is computed.

1. A desired set of vertices $D : \{d^1, ..., d^R\}$, that should remain in the old plan during replanning, is computed.
2. Given a desired set D, G_{ADG} is reversed to create a new graph G^R_{ADG} that contains the same vertices for every agent i up to d^i. Existing edge connections for these vertices are included in G^R_{ADG} with reversed direction.
3. With G^R_{ADG} , for every agent i starting from the desired vertex d_i,G^R_{ADG} is traversed and all vertices that can be reached from d_i are added to the commit cut c_i. The commit cut C is found by taking the latest reachable vertices for all agents.

Choosing a desired set requires domain knowledge. In general replanning is triggered if one agent comes close to finishing its current plan. A desired set of vertices should be chosen such that the expected time differences agents will finish their last action in the old ADG is minimal. Otherwise this could cause idle times for some agents if they do not have any more actions in the old plan and their first new plan actions depend on the old plan actions of other agents. For more details on how this works, the reader is referred to the original paper: (Hoenig *et al.*, 2019).

2.6.5.2 Example

To illustrate the concept of the ADG framework, let's revisit the scenario depicted in Figure 4. Both agents A_G and A_B want to switch positions and avoid each other in the process. Figure 4 depicts a potential solution to this problem, taking **Following Conflicts** into account (see Chapter 2.3.1.2 (MAPF - Conflicts) for reference). One potential solution, depicted on the right side of Figure 4, is to let A_G move to the avoidance area and move out of the way for A_B. To be more specific this corresponds to the following MAPF plan: $\Pi = (\pi_G, \pi_B)$ with

$\pi_G = \{\text{move}(A, B), \text{move}(B, D), \text{wait}(D), \text{wait}(D), \text{move}(D, B), \text{move}(B, C)\}$

$\pi_B = \{\text{wait}(C), \text{wait}(C), \text{move}(C, B), \text{move}(B, A)\}$

While this plan is sound, directly executing this commands on AMRs can still result in a collision if, for instance, moving from C to B takes A_B significantly longer then A_G waits at D. The ADG framework solves this by converting plan Π into an action precedence graph G_{ADG} as described in Algorithm 5. The resulting graph G_{ADG} for plan Π looks as as depicted in Figure 11 below.

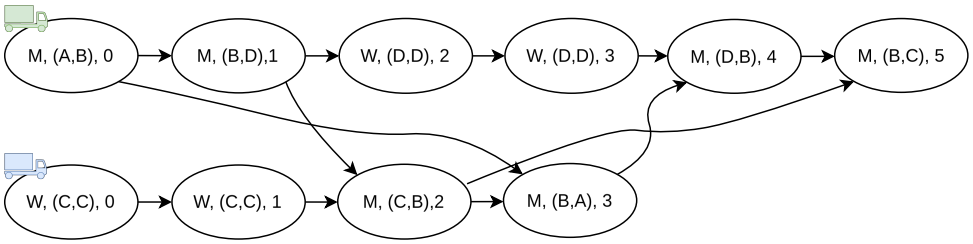


Figure 11: Initial ADG state after being created from MAPF solution Π. M, W represent move or wait actions. (X,Y) are start and goal nodes. The right most number is the time step in the original MAPF plan. A directed edge between two vertices p^i and $p^{i'}$ indicates that action $a^{i'}$ can only be executed after action a^i has been completed.

After G_{ADG} is created, **enqueue_vertices()** is called for each agent i to set the initial state of the vertices. In this example, this leads to the following initial state of the ADG depicted in Figure 12 below. Yellow vertices are in state *enqueued* and ready for execution, grey vertices are in state *pending* and cannot be executed yet.

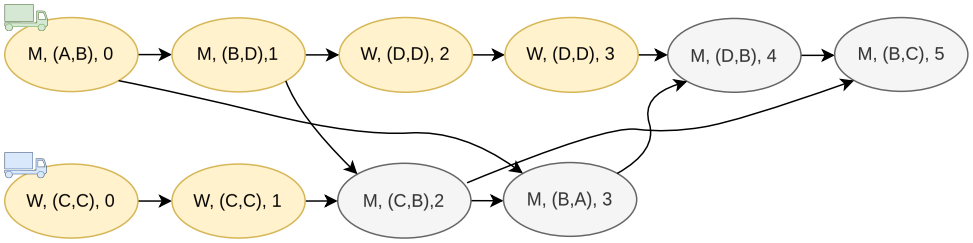


Figure 12: State of an ADG after vertices have been enqueued. All vertices in state *enqueued* (ready for agent execution) are yellow. Pending vertices are grey.

After initialization, agents start executing all actions available to them (state *enqueued*) in the order they arrived in their action queues q^i. Figure 13 shows the ADGs state after A_G has finished its first two actions and A_B its first one. ***Note*** that after A_G finished its second action, the third and forth action for A_B were set to *enqueued* (yellow). Adhering to this action precedence ensures that the original MAPF plan is executed correctly, even if individual actions take longer then planned.

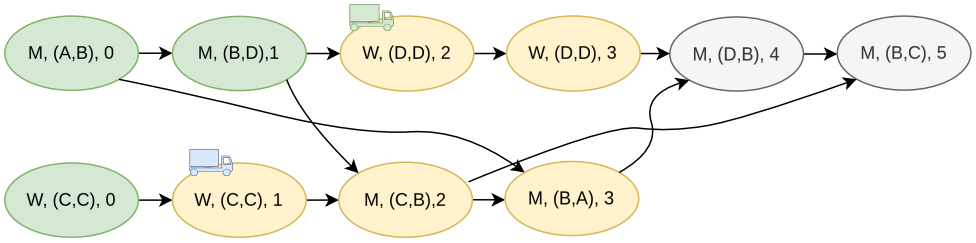


Figure 13: ADG state after some actions have been completed. The green agent A_G has *completed* its first two actions (green). After notifying the system the third and forth action for A_B were set to *enqueued* (yellow). Meanwhile A_B has finished its first action and is working on its second one. The last two actions for A_G are still *pending* (grey) and will be *enqueued* after A_B finished its last action.

2.6.6 Planning for a windowed time horizon (RHCR)

Despite recent progress solving the MAPD problem remains challenging. One often used approach is to treat its lifelong nature as a series of sequential MAPF problems. While this works, there are two main drawbacks to this setup:

1. It is not well defined how agents that already finished their current task and do not have a new task yet (e.g. are idle) should behave.
2. Whenever an agent receives a new task a full replanning of the whole system is necessary and therefore a lot of redundant planning occurs.

To address these issues, (Li *et al.*, 2021) proposed the Rolling Horizon Collision Resolution (RHCR) framework. In this context lifelong MAPF planning works as follows:

Instead of planning for all agents from their start to their respective goal location, and avoiding collision for that full duration, agents only avoid collisions with each other for a windowed time horizon w. Replanning is done every h time steps. Both parameters h and w are user-defined but to ensure collision free paths, they are chosen such that $h < w$. This ensures that replanning is done before the current planning horizon ends. In this context, the planning effort directly depends on the planning horizon w. The authors showed that choosing a small w significantly speeds up planning time, while usually not degrading in solution quality. To mitigate problem 1 and 2, the authors suggested assigning multiple tasks to agents such that every agent is always busy, longer than the planning horizon w. Together with the RHCR framework, this multi-task assignment that continous replanning is well defined and only has to be done every replanning period h.

While using RHCR looses theoretical guarantees, in practice it is often faster and more robust than traditional lifelong MAPF planning. To mitigate the theoretical deadlock problem that can occur for paths longer then w, the authors suggest an adaptive w depending on if shuttles made sufficient progress towards their goal. If not w should be iteratively increased until the deadlock is solved.

Another significant advantage of using RHCR for lifelong planning, is that planning does not break down if no valid solution exists or the planner can not find one. Using RHCR in such cases allows the system to continue operation even if, for instance only some agents are able to make progress towards their goal and others are only able to wait on the spot. In a lifelong context, such situations can usually be resolved once blocking shuttles have finished their tasks and move to other areas. There, planning on the full planning horizon would lead to an undesired planning failure. For further clarification, consider the example depicted in Figure 14.

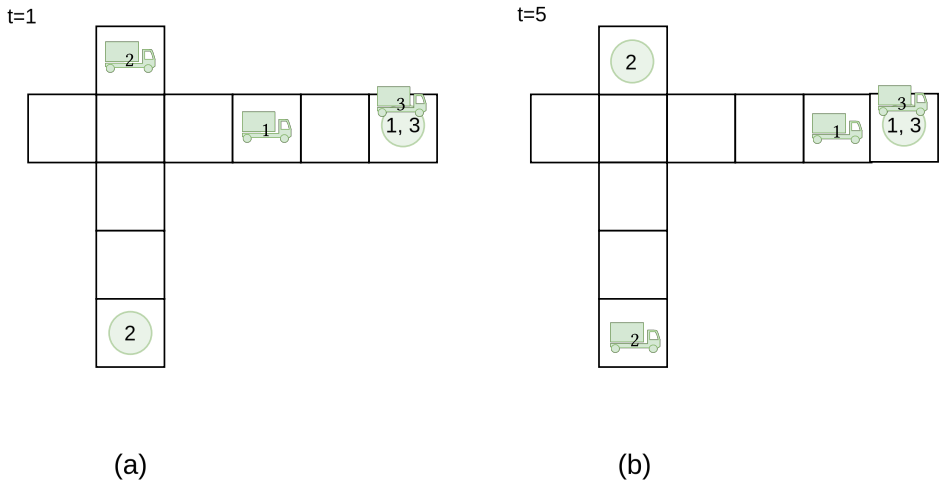


Figure 14: A lifelong MAPF problem with 3 agents (1,2,3). Using the RHCR framework allows progress to continue even if two agents are stuck. (a): The initial plan. (b): Four time steps later.

Three agents (A_1, A_2, A_3) collaboratively finish warehouse pick and place tasks. A_3 has just moved to its goal node but due to an unexpected delay it has not started its next task yet. Therefore it happens that in the next planning round, agent A_3 and A_1 have the same goal node. In the strict MAPF sense, this corresponds to an invalid plan and would make any planner fail. However, using a planning window, A_1 will only consider collisions up to a planning horizon w and overall planning would still succeed even though for A_1 this means waiting at its current position until the end w. This allows other agents (A_2) that are not involved in the commotion to receive normal planning results and continue operation. Even without RHCR, A_3 would eventually continue to move to another goal, and operation would continue normally, but only after a significant delay for all shuttles that are not involved in the commotion.

2.6.7 The RHCRE framework

In (Varambally, Li and Koenig, 2022) the authors propose a lifelong MAPF framework, that solves the problem of continous replanning (as discussed in Chapter 2.6.6 (Planning for a windowed time horizon (RHCR))) as well as robust execution of plans on agents with real robot dynamics (see Chapter 2.6.3 (The Action Dependency Graph (ADG) framework)). The Rolling

Horizon Collision-Resolution and Execution (RHCRE) framework does this by combining the RHCR and ADG concepts into one comprehensive Multi-Agent Pickup and Delivery (MAPD) framework that is directly applicable to be used in warehouse routing.

2.6.7.1 Conceptual overview

Conceptually it works as follows: Planning is done like in the RHCR framework. Paths are planned for all agents, while collisions are resolved only up to windowed time horizon w. In the RHCR framework, all agents are assigned multiple tasks longer then time horizon w. Together with periodic replanning RHCR ensures continuous operation. Because the RHCRE framework uses an ADG for plan execution and finding out when replanning is necessary, it does not need to assign multiple tasks to agents ahead of time, nor is periodic replanning required. In the context of ADGs, agents notify when they are a specified amount of time away from finishing their goal. This is equivalent to a dynamic time horizon w. Another advantage of using ADGs allows for the overlapping of planning and execution, further mitigating agent idle times. The RHCRE framework thus incorporates all advantages of RHCR and ADG.

Figure 15 highlights that the RHCRE framework can be directly used in a warehouse routing system. During planning new tasks for every agent are received from an external task allocator (violet colored tile). Then planning is done by only considering collisions for the next w time steps. The found plan (using any sequential MAPF planner) is converted into an Action Dependency Graph (ADG). With this ADG, agents continuously work on their action queue and receive new ones until one agent only has n actions left. Then the ADG's commit cut is used to provide start positions for the next MAPF planning round. After fetching new tasks for all agents, planning is restarted and the found ADG is concatenated to the end of the old one. For further details, the reader is referred to Chapter 2.6.6 (Planning for a windowed time horizon (RHCR)), Chapter 2.6.3 (The Action Dependency Graph (ADG) framework) and their related papers.

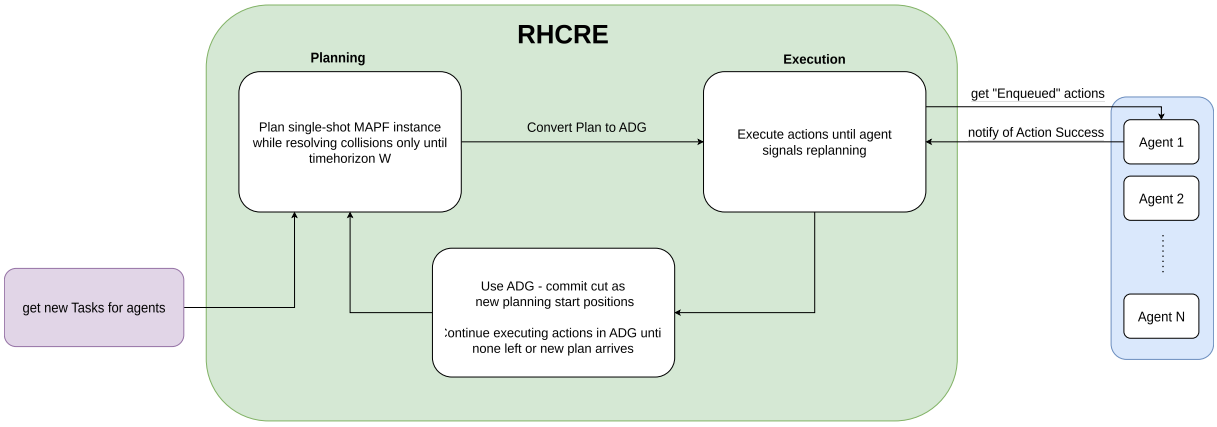


Figure 15: Conceptual diagram of the RHCRE framework in a warehouse routing system. The figure illustrates how the RHCRE combines rolling horizon planning with advantages of the ADG framework to ensure continous operation in a warehouse setting.

3 Fleet Control System (FCS)

This chapter provides an overview of the Fleet Control System (FCS) system, its inner workings and its role as the central component for multi-fleet coordination in warehouse systems deployed by KIN. Chapter 3.1 (Definitions (FCS)) discusses basic concepts and terminology. As this thesis aims to propose a new centralized routing strategy to be integrated into FCS, special emphasis is placed on analyzing the current routing system and its components, discussed in Chapter 3.3 (Path Planning and Fleet coordination). Subsequently Chapter 3.4 (Drawbacks and Mitigation strategies) will address challenges faced by the existing decentralized fleet coordination system, followed by a discussion on implemented mitigation strategies.

For a more comprehensive review on the inner working and communication mechanisms of FCS the reader is referred to (Kohout, 2024).

3.1 Definitions (FCS)

This section discusses key terms and concepts necessary to understand the FCS.

3.1.1 Shuttle

Shuttles also referred to as AMRs or agents, are central to the FCS. Their responsibilities include navigating from one station (Chapter 3.1.2.2 (Station)) to another, picking up and delivering goods within a warehouse. To do this, AMRs plan their paths and navigate over a shared road network (Chapter 3.1.2 (Lanemap)). The FCS is designed to be compatible with any AMR, allowing for the seamless integration of shuttles with different sizes and capabilities into the same warehouse system (Chapter 3.3.2.1 (Shuttle Types and their Footprints)).

3.1.2 Lanemap

Lanemaps represent the shared road network that shuttles navigate on. A Lanemap is a graph representing the warehouse layout. It consists of goals (Chapter 3.1.2.1 (Goal)) and lanes (Chapter 3.1.2.3 (Lane)) that connect them. During warehouse operation, shuttles receive orders (Chapter 3.2.1 (Order Hierarchy)) instructing them to move from one station to another (Chapter 3.1.2.2 (Station)).

Consider the example depicted in Figure 16. The yellow dots are goals, and the connecting arrows are lanes. Left and right there are stations (indicated by small rectangular icons) where shuttles can pick up and deliver goods. In this 2 shuttles system, goods are transported from the left side of a warehouse conveyor belt system to the right side, essentially connecting two different production lines. The middle area is defined as a round course where two shuttles can go back and forth between the stations. Two turnable goals (indicated by two circular arrows) connect the round course with the station areas.

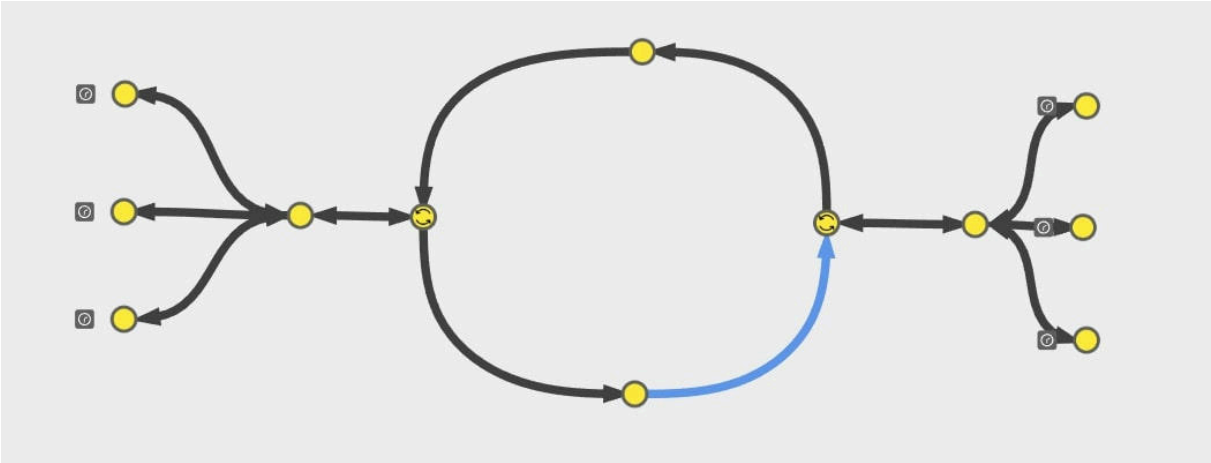


Figure 16: A simple Lanemap in FCS. Yellow dots are goals and connecting arrows are lanes. Two circular arrows on a goal indicate turning on the spot is allowed. Small rectangular icons next to a goal indicate a station.

3.1.2.1 Goal

Goals are nodes in the Lanemap, representing 2D positions in space. Goals have properties that control how shuttles interact with them.

- **Turnable**: If a goal is turnable rotation on the spot is allowed.
- **Stations**: Goals can contain a list of stations associated with them.
- **AnchorPoint**: If a goal is marked as an anchor point, shuttles are not allowed to stop there and can just drive through.

3.1.2.2 Station

Stations are locations in warehouses where shuttles can pick up or deliver goods. To drive to a station a shuttle first navigates to its associated goal and then performs a predefined fine-positioning routing to the actual location of the station were logistic interactions take place. These fine-positioning routines can be quite complex, and are part of the shuttle's internal software stack. In the scope of this thesis, these fine-positioning routines are not considered. Instead, we assume that the FCS only distributes orders (Chapter 3.2.1 (Order Hierarchy)) to shuttles, which they must execute independently.

3.1.2.3 Lane

Lanes are modeled as clothoid curves connecting goals. They directly represent the paths shuttle follow during navigation. A lane's direction $(\leftarrow, \rightarrow, \leftrightarrow)$ defines which direction a shuttle can travel on it. (e.g. in Figure 16 the central round course can only be traversed counter-clockwise, while the bidirectional connection lanes can be traversed in both directions). This fine granularity allows for modeling of complex driving behaviors and efficient coordination depending on physical warehouse constraints.

Since lanes are modeled using clothoid curves, smooth transition during and between lane traversal is possible. This is the case because clothoid curves are a special type of curves that always maintain a linear change of curvature. This ensures for smooth acceleration patterns during lane traversal. There are multiple types of clothoid representations. Because G1-Hermit clothoid are solely parametrized by their start and end points and in and out degree, they naturally fit well into this design. Therefore lanes also store in and out angle (in degree) for their goal connections. For further information on clothoid curves the reader is referred to resources like (V.P. Kostov, 1992) or (Scheuer and Fraichard, 1997). Clothoid curves are very useful in roadmap construction as they ensure smooth transitions between lanes if their out and in orientations match. In the context of FCS, if e.g. on a goal the in and out degree of consecutive lanes match, a shuttle can move over the goal without stopping because no on the spot rotation is necessary.

Figure 17 depicts the two possible ways shuttles can move between goals. On the left side the in and out degrees of consecutive lanes match $(0° \rightarrow 45°), (45° \rightarrow 90°)$, on the right the shuttle has to rotate on the spot and the middle goal therefore has to be marked as turnable. In the right example if the middle goal was not indicated as turnable, then a shuttle could not transition over it.

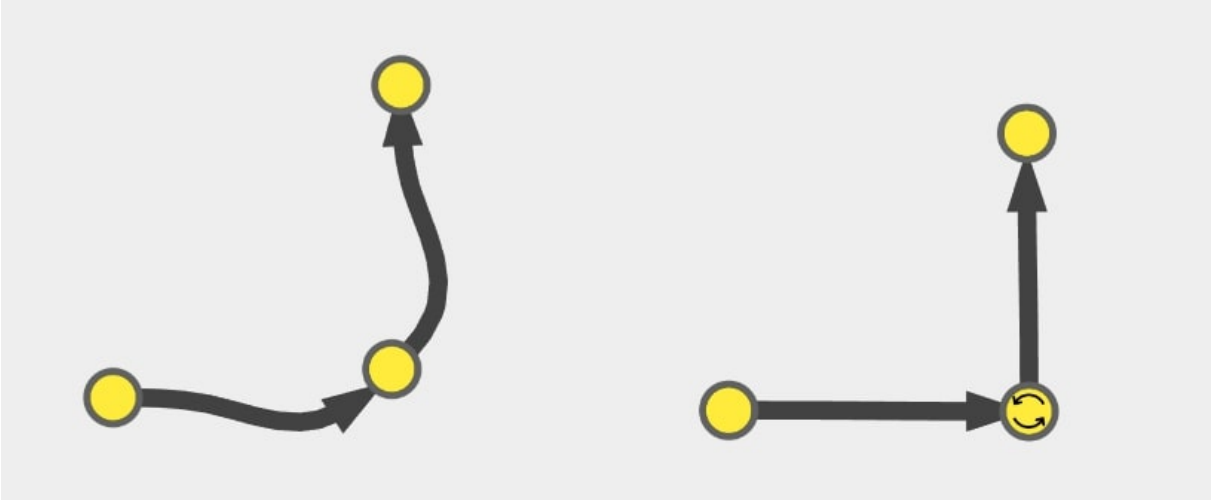


Figure 17: The two possible ways shuttles can move between goals in FCS. (a): The lanemap allows for continous motion because lane orientations match. (b): A lanemap where turning on the spot is necessary.

3.2 Order Management

Besides coordinating shuttles through its Lanemap, the FCS is also responsible for taking in orders from external dispatch-systems and distributing them to its shuttles. In the warehouse context, whenever an object needs to be moved from A to B, an order is sent to the FCS.

3.2.1 Order Hierarchy

In essence, an order is a sequence of load, unload or shuttle specific tasks (like checking an area). Every FCS task represents multiple simple executable motion based tasks for the shuttle itself (Shuttle Task). Figure 18 shows this hierarchical relationship.

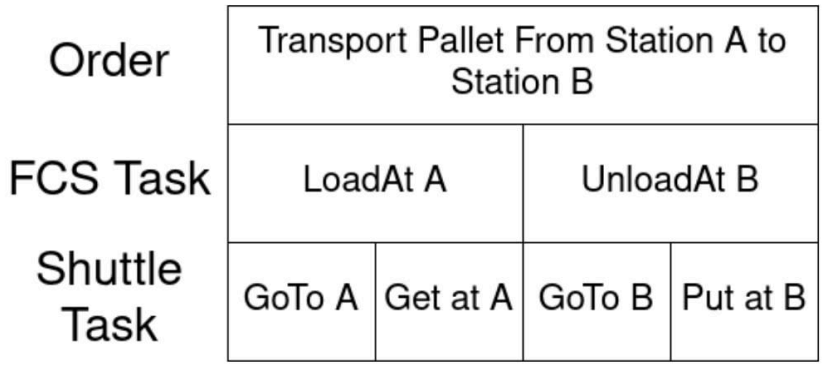


Figure 18: The order hierarchy in FCS. Orders contain multiple FCS tasks. Each task represents multiple shuttle specific actions.

3.2.2 Order and Task Allocation

In traditional literature, task allocation problems were solved independently of routing, because a shared formulation was considered too complex (Pinedo, 2008). The FCS follows this design. In the FCS, the order allocation is done as follows:

1. On startup, a cost-map between all goals is precomputed. ***Note*** that lane traversal costs are extracted from the graph-level Chapter 3.3.1 (Routing Graph).
2. During operation, whenever a shuttle finishes its order, it is assigned a new order (from all available ones) based on which order has the lowest cost to execute. This is done by calculating the costs of the order's task stations based on the aforementioned cost-map.

3.3 Path Planning and Fleet coordination

This section will go into detail about how planning, reservation and shuttle coordination is currently done in FCS.

3.3.1 Routing Graph

The Lanemap represents the road network shuttles navigate on (see (Chapter 3.1.2 (Lanemap)). It consists of lanes (clothoid curves) and connecting goals (Positions in Space). As already described, how shuttles can interact and move over goals and between lanes depends on the in-and out-degrees of lanes and goals turnable property. This higher order concepts do not allow for standard graph based path planning. Therefore, FCS uses a 2 layered approach. Underneath the lanemap level a routing graph exists that captures these complex (orientation-based) transitional relationships between lanes and goals in a more fine-grained manner. On

this underlying graph, the actual algorithmic path planning and agent coordination is done. Throughout this thesis, this will be referred to as **routing graph** or **orientation graph** interchangeably.

To further understand the differences between the Lanemap and the routing graph level, consider Figure 19:

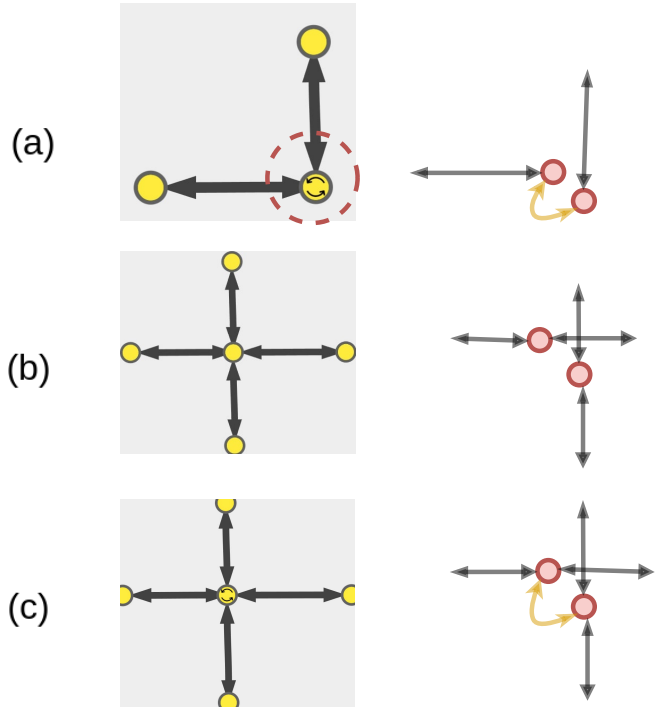


Figure 19: Comparison of a lanemap (left) and its underlying routing graph representations (right). (a): A simple turnable corner goal, (b): An intersection where rotation is not possible, (c): An intersection where rotation is possible.

On the left side various lanemap structures are depicted, on the right side their underlying routing graph representations. From top to bottom:

1. A simple turnable corner goal. Its routing graph representation contains one node per possible orientation a shuttle can have when staying on that goal or traversing it. The yellow edges represent orientation transitions between them. This fine-granular representation of orientation transitions ensures that path-planning directly corresponds with the real path a shuttle would take on the lanemap.
2. An intersection where rotation is not possible. Depending on the physical structure of the warehouse layout, rotation on the spot may not be possible for larger shuttles. In this scenario, shuttles can only go from left to right and top to bottom but can not move around the corner. Since the routing graph does not contain any orientation transitions, no such path will be planned either.

3. An intersection where rotation is possible. In this case, shuttles can also move around the corner because the intersection goal is marked as turnable and its routing graph therefore contains necessary orientation transition-edges.

3.3.1.1 Detailed Example

Note that Figure 19 above provides a simplified example of the routing graph layer. Figure 20 gives a further detailed example of an actual routing graph layer exported from FCS.

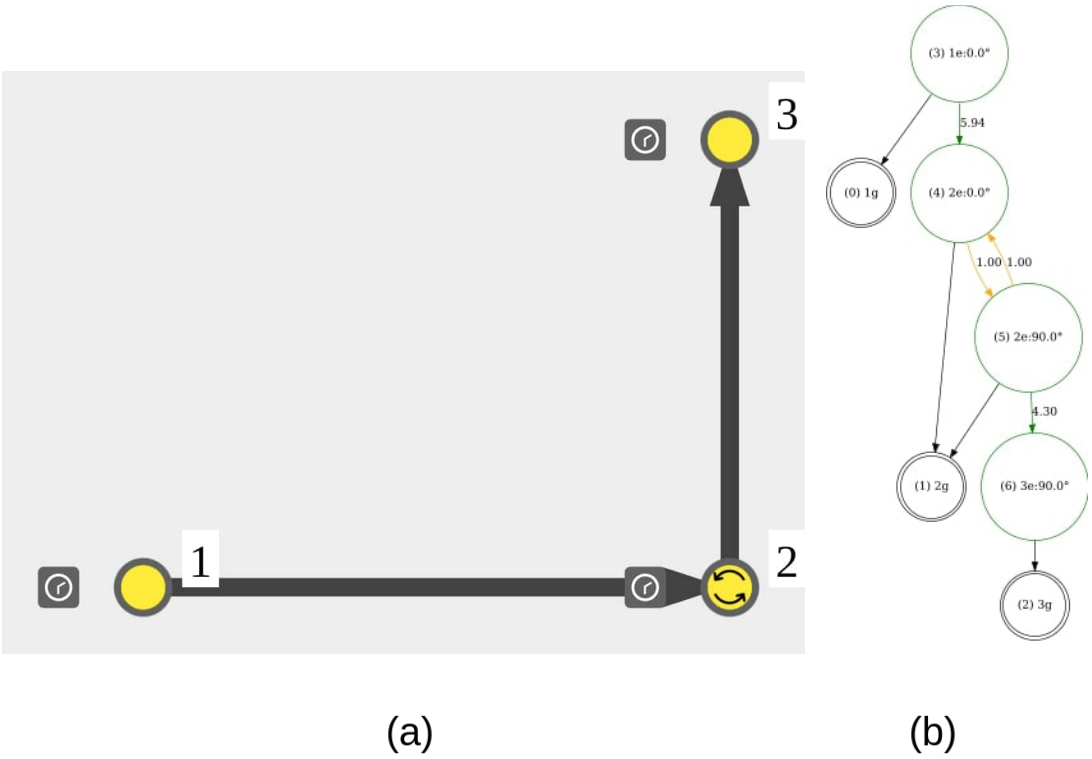


Figure 20: Detailed comparison of a lanemap (a) and its underlying routing graph (b) exported from FCS. Green nodes represent goal nodes, double circled nodes represent goal stations (related to FCS-tasks). Parenthesized numbers in the nodes labels represent the nodes unique ids in the routing graph layer.

There, the lanemap (left) contains three goals (1,2,3). All three have a station attached to it. Goal 2 is turnable allowing for a transition from goal 1 to 3. Corresponding routing graph (right) contains one node (goal and transitional orientation pair (green circles)) for each goal 1 (1e:0.0°) and 3 (3e:90.0°). Goal 2 has two corresponding nodes (2e:0.0°, 2e:90.0°) because it is turnable and therefore can be traversed in 0.0° and 90.0° orientation. Orientational transition edges (orange) indicate this. Because each goal has a station associated with it, the planning graph contains station goals (1g, 2g, 3g) indicating a terminal planning node (double circles). In FCS, costs associated with edges represent the distance between goals. Moreover lanes can have additional (manually defined) costs associated to them. These costs are added to the overall edge costs. Orientational transitions have an angle dependent cost that is calculated based on the smallest angular distance between two orientations.

In Figure 20, (2e:0.0°, 2e:90.0°) are 90° apart and the cost for this transition is, e.g. 1.0.
This costs of course, do not model shuttle kinodynamic properties accurately, but ensures that shuttle prefer paths where continous navigation without in-place rotation is possible.

3.3.2 Reservation Grid

During FCS operation multiple shuttles move over a shared lanemap. To ensure that shuttles do not collide, a reservation grid is used. This reservation grid works as follows:

1. Physical space is discretized into an occupancy grid like structure. (Moravec and Elfes, 1985).
2. For every node and edge (entity) in the orientation graph, a polygon of the size of the shuttles footprint is moved over this edge or positioned at that node and every cell that is overlapped by that polygon is marked in the occupancy grid.
3. For each possible entity a in the graph its overlap relationship $O_R(a)$ is built by checking with which neighboring other entity b it geometrically overlaps based on shared occupancy grid information.

Consider the following simple example: if e.g. a shuttle i travels over an edge $(X \rightarrow Y)$, it would geometrically overlap at least another shuttle staying at X or Y. Given no other nodes or edges close by, the overlap set for edge $(X \rightarrow Y)$ would be: $O_R((X \rightarrow Y)) = \{X, Y\}$
During operation a shuttle i follows its shortest path π_i, and reserves it by pushing its path's overlap information $O_R\left(\pi_{i[1...n]}\right)$ into the reservation grid. With path π_i reserved, other shuttles will reserve their own path as follows: Another shuttle k with a path π_k may try to allocate its own path π_k. Let path π_k contain nodes $[v_1, ..., v_n]$ a path segment π_k^s is a subset of consecutive nodes $[v_i, ..., v_j]$. During path reservation, k will start with the first node v_1 and check if it can reserve it by checking if all overlapping entities in $O_R(v_1)$ are free. This is done sequentially for as many nodes in π_k as possible. After the longest possible segment π_k^s is found, k will reserve it by pushing its overlap information $O_R(\pi_k^s)$ into the reservation grid. This process is done for all shuttles in the system. Whenever any shuttle releases parts of its paths, this process is repeated.
Chapter 3.3.3 (Employed Decentralized Fleet coordination) describes how this simple strategy works because of the asynchronous nature of the FCS's routing system.
Figure 21 shows a simple example of a reservation grid based navigation. The blue shuttle wants to move to its station goal. Meanwhile violet performs a task at its goal. Because the reserved cells for both paths do not overlap, the blue shuttle can proceed without waiting.

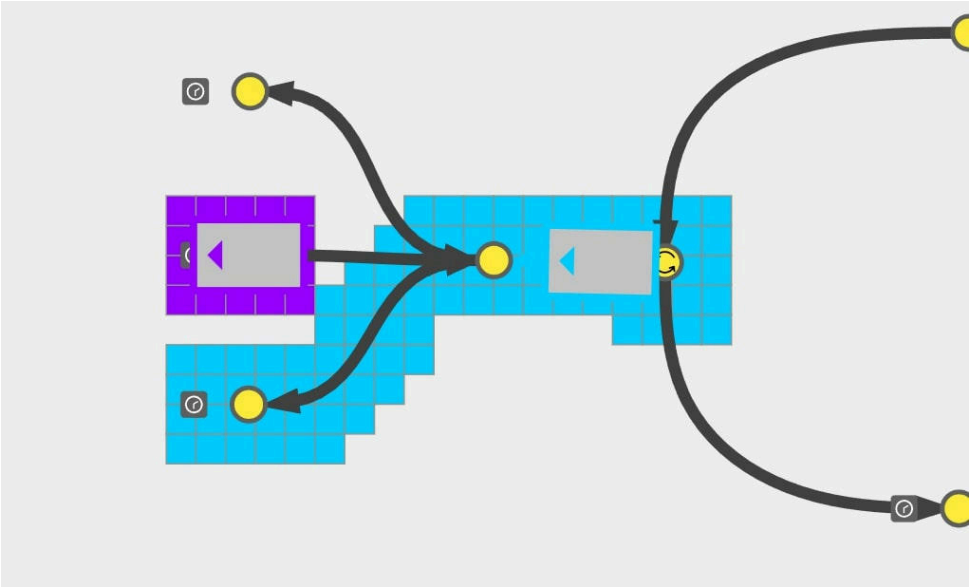


Figure 21: Illustration of the reservation grid used in FCS. The blue shuttle reserves space for its path moving over the lanemap, The violet shuttle is far enough away so that the blue shuttle can proceed.

3.3.2.1 Shuttle Types and their Footprints

As mentioned before, in the FCS, geometric overlap relationships O_R are computed depending on a shuttles footprint. The aforementioned overlap relationship strategy works for shuttles of any size. This way the FCS can be easily configured for new shuttle types.

3.3.3 Employed Decentralized Fleet coordination

In FCS, multiple shuttles navigate over a shared Lanemap. To ensure safe collision free navigation shuttles reserve path segments in the reservation grid (Chapter 3.3.2 (Reservation Grid)). Finding paths for shuttles is done in a decentralized manner. This means that every shuttle plans its path independently. It works as follows:

In the beginning no shuttle has a shuttle task (Chapter 3.2.1 (Order Hierarchy)) assigned to it. When a shuttle i receives a new task, an A* - like search algorithm is done over the routing graph (Chapter 3.3.1 (Routing Graph)). This resulting path π_i (a sequence of 1 to n nodes and edges), represents the shortest path from i's position to its goal station. Then i tries to allocate a predefined number of goals k in π_i with $k \ll n$. During operation, shuttles follow their path segments $1..k$ and reserve them in the reservation grid. Often, they may not be able to reserve any nodes or edges in their path segments for themselves because other shuttles reserved them first. In that case they go into idle mode.

Whenever a shuttle makes progress towards its goal and releases a previously held node, all other shuttles are woken up and try to reserve their own path segment again. If two shuttles want to reserve the same goal, it is randomly decided which one comes first. This simple scheme works due to its asynchronous nature.

Figure 22 shows the above described architecture in detail. ***Note*** that *FCSNav* refers to the underlying graph and path planning stack inside FCS.

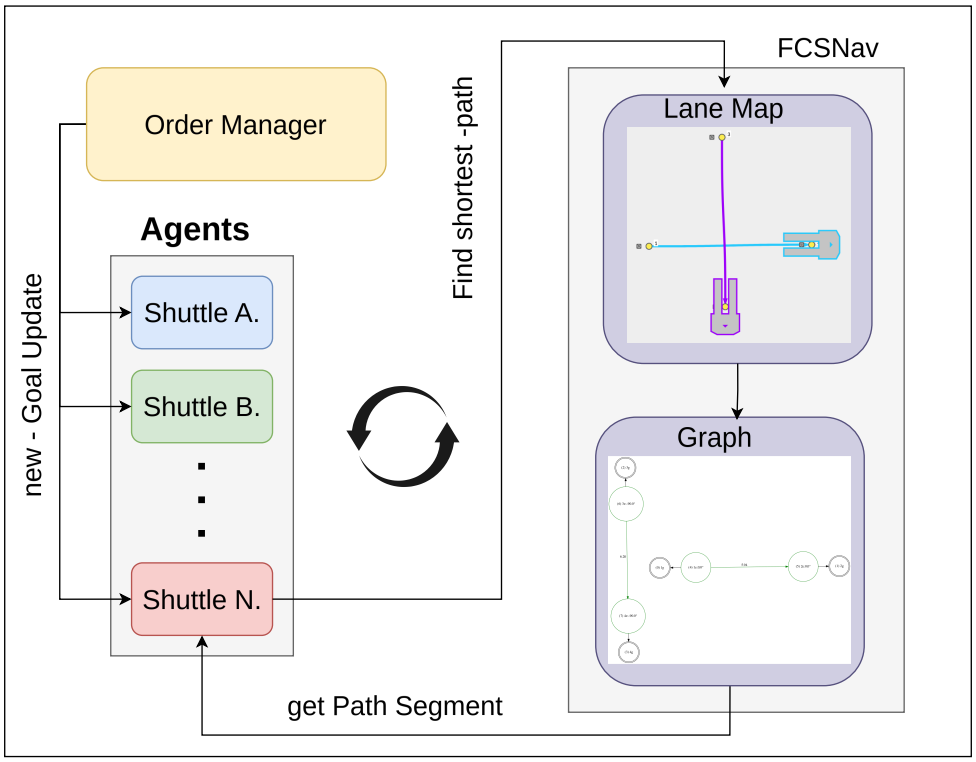


Figure 22: The decentralized fleet coordination strategy used in FCS. Shuttles independently plan their path and follow path segments towards their goal.

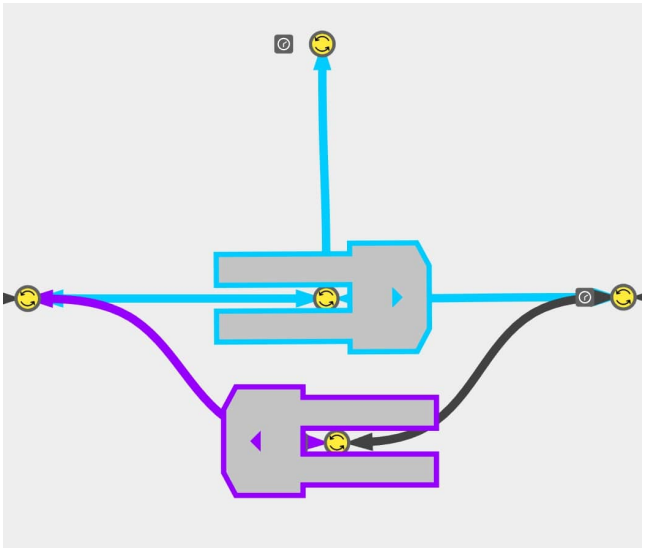
3.4 Drawbacks and Mitigation strategies

The above mentioned decentralized strategy is simple, fast and scalable. However, its robustness heavily depends on the structure of the used Lanemap. If, e.g. the Lanemap represents a round course where shuttles can never face each others path head on then it works perfectly. Unfortunately, some warehouse layouts are too tight or crowded for this, and require narrow bidirectional corridors. There, the current scheme can lead to congestion or deadlocks, if e.g. two shuttles try to move into such a corridor from opposite directions. Since Lanemap structures can be arbitrarily complex, deadlocks can occur in all kinds of situations. Figure 23 depicts two of these situations most often faced by FCS. From top to bottom:

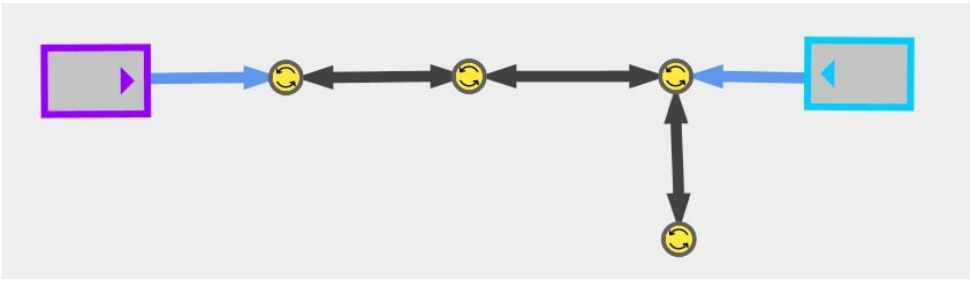
1. An avoidance area deadlock. The teal shuttle S_v started from the left and had an order to move to the right. The violet shuttle S_v, planned to move from right to left. In this situation S_t is first send to the center goal, then S_v is send to the avoidance area below. Then S_t is manually triggered to continue moving right and operation proceeds as normal. If however, while S_t performs the avoidance routine with another shuttle, its order is canceled, and a new one is send to pickup an item at the goal on the top, then it is blocked from its path

by S_v because it can not rotate. S_v on the other hand can not continue because it would physically collide with S_t while traveling over the left curve.

2. Two shuttles S_v and S_t face each other head on in a narrow corridor. Theoretically, S_t could move out of the way into the small corner goal, but this would require agents to be planned collaboratively.



(a)



(b)

Figure 23: Two deadlock situations that can occur in FCS. (a): A rotational deadlock caused by sudden order changes. (b): Two shuttles face each other in a narrow corridor

To mitigate the above mentioned issues, elaborate strategies have been implemented in FCS. These are the most important ones:

3.4.1 Manually designed avoidance areas

As seen in the top picture of Figure 23, the FCS system allows for the manual design of collaborative shuttle maneuvers across specific sets of goals. As already described in Chapter 3.4 (Drawbacks and Mitigation strategies), when two shuttles arrive at two specific goals, a manual avoidance subroutine is triggered that manually sends move commands to shuttles in a specific order. After, that operation continues normally. This is a simple and often effective

way to avoid deadlocks if there are specific areas in a warehouse Lanemap that are especially prone to cause problems. Unfortunately, this is a labor-intensive manual process that requires a lot of domain knowledge and does not generalize well.

3.4.2 N-Robot area

N-Robot areas are another manual deadlock mitigation strategy. An N-robot area defines a set of goals where no more then N shuttles are allowed to be at the same time. This is quite versatile and can be used to decrease traffic in critical areas, or if N = 1 to avoid deadlocks. Figure 24 shows an example of an N-robot area in a warehouse layout. A two way highway like roadmap leads through a broader warehouse area. In the middle there is a bidirectional connection that allows shuttles to switch lanes. If however two shuttles want to use this connection at the same time, a deadlock would occur. To avoid this, the connection area is defined as an N(1)-robot area. This way one shuttle can always pass through the connection area without being blocked by another shuttle.

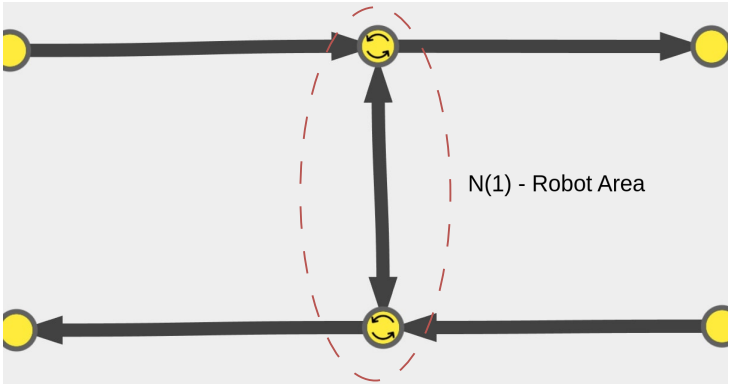


Figure 24: An N-robot area used in FCS to avoid deadlocks. A two way highway like roadmap leads through a broader warehouse area. In the middle there is a bidirectional connection that allows shuttles to switch lanes.

N-Robot areas are a quite powerful tool for manual Lanemap annotation, but they are also labor-intensive and require a lot of domain knowledge to setup properly. While Figure 24 shows how an N-Robot area can be used to avoid deadlocks, it comes with a slowdown in overall throughput, because a shuttle moving over the top highway lane will interfere with the path of another shuttle moving over the bottom lane, even if none of them plans to change lanes.

3.4.3 Deadlock resolution

In addition to the above mentioned manual Lanemap annotation strategies, FCS also features a reactive deadlock resolution strategy. It works as follows:

When a shuttle S_v can not reserve its path-segment p_v^s because another shuttle S_t already reserved it, it goes into idle mode. An edge $(S_v \rightarrow S_t)$ is then added to the shuttle dependency graph. This graph is a directed graph where nodes represent shuttles and edges represent de-

pendencies between them. When two or more shuttles are in a deadlock situation a circle in the graph exists. Therefore whenever a shuttle adds itself as a dependency, the graph is checked for cycles. If a cycle is found, a deadlock is detected.

If a deadlock is detected in this manner, one of the involved shuttles is randomly sent to a random free goal nearby. This reactive strategy resolves deadlocks but can lead to suboptimal routing behavior (shuttles will take much longer paths) and is not guaranteed to work in all situations. This is the case because some situations lead to a live lock. In such a situation two or more shuttles are randomly sent around by the deadlock-resolution strategy, but do not make progress towards their goal and go back and forth (seemingly) indefinitely.

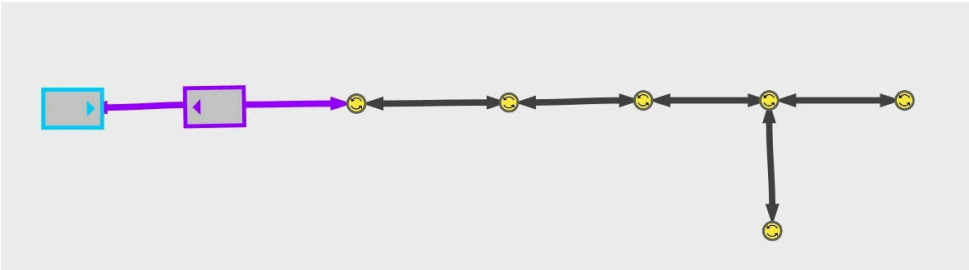


Figure 25: A life-lock situation in FCS. Two shuttles face each other in a narrow corridor and deadlock resolution can cause shuttles to go back and forth.

Figure 25 shows a simple example of a life-lock situation. Two shuttles S_v (violet) and S_t (teal) face each other head on in a narrow corridor. Since reserved shuttle paths depend on each other, a deadlock is detected. Deadlock resolution then sends one of the shuttles to another nearby free goal at random. In the first round only S_v can be sent away. Then either S_v or S_t move into the now free space. In this scenario the life-lock would only be resolved if S_v randomly drives to the avoidance goal.

Theoretically, a life-lock will eventually be resolved due to random shuttle behavior. However, this is unreliable and can take an unpractical amount of time for warehouse operation.

One common approach used by FCS in such narrow corridor situations is to place an N(1)-robot area across the whole corridor, ensuring that only one shuttle can move through it at a time.

Overall the FCS has evolved into a quite powerful and robust system, capable for allowing robust fleet coordination. However, it is still heavily dependent on manual Lanemap annotation and reactive strategies that both lead to suboptimal throughput.

Instead of decentralized driving and reactive deadlock resolution, another approach could be to collaboratively plan shuttles from the beginning. This is the approach taken by this thesis.

4 Design

To build a scalable warehouse routing system directly applicable in FCS, a design is developed to work with already existing components. Because of the complexity of FCS and its tight coupling with some features not directly relevant to this Proof-of-Concept, it was decided to not implement the new routing system directly in FCS, but rather in a separate bare-boned simulation environment that closely mimics how FCS works. This allowed for a more focused incremental development and testing process necessary when building such a complex system. Chapter Chapter 4.1 (Simulation (FCS Fake)) will go into further detail on the simulation environment and its differences to the real FCS system.

In FCS, an arbitrary number of shuttles collaborate on a shared roadmap graph, to fulfill orders (sequences of pick and place tasks) continuously. New orders may arrive at any time. Because FCS uses a decentralized system (see Chapter 3.3 (Path Planning and Fleet coordination), it is robust to individual shuttle failures and can scale to a large number of shuttles. The centralized system developed in this thesis is specifically designed to ensure that the same properties still hold while being more robust to deadlocks and better scale to larger numbers of shuttles.

Since this centralized routing system should replace the one currently used in FCS in the future, it was specifically designed with modularity in mind. This should allow to swap out implementations of individual components incrementally until the full system is integrated into FCS.

Figure 26 shows a high-level architectural overview of all components used in the (MAPD) routing system as well as how they integrate into the simulation environment. Information from the original *FCSNav* is used to initialize the routing system with the existing roadmap graph and the reservation grid information. An Order Manager (OM) generates orders (sequences of tasks) and communicates their status with the Shuttle Manager (SM). Agents communicate with the Action Dependency Graph (ADG) about their progress and when new actions are available. In the MAPD routing system, the SM functions as the main communication hub, for all agents and the OM. It decides when replanning is necessary and requests new plans from the MAPF planner.

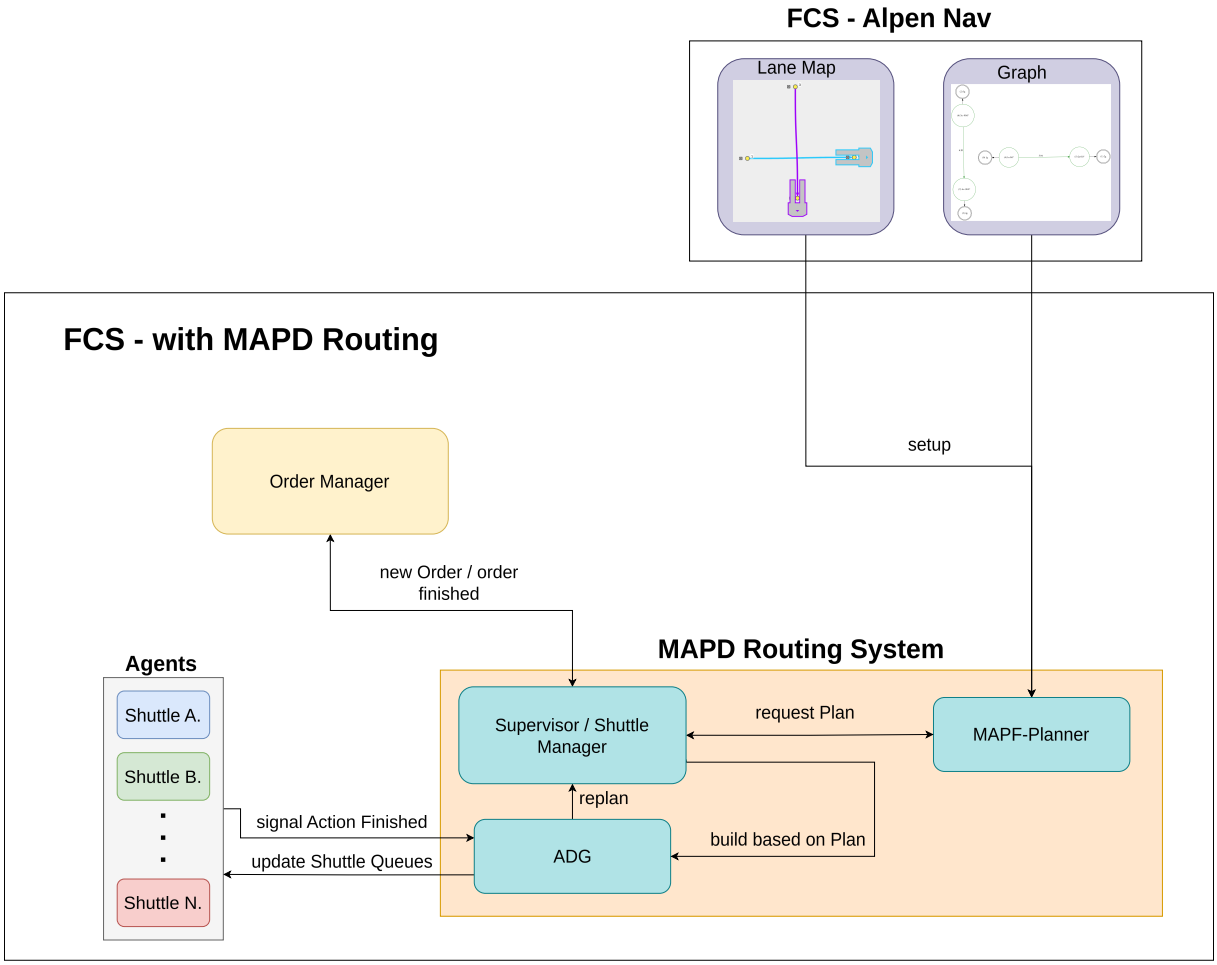


Figure 26: Overview of FCS with the new MAPD routing system. The figure depicts the interaction between the Order Manager, Shuttle Manager, and MAPF-Planner within the MAPD routing systems architecture.

The following sections will go into further detail on each of these components.

4.1 Simulation (FCS Fake)

To ensure incremental development and testing, a bare-boned simulation environment that closely mimics how FCS works, was developed (see Chapter 3 (Fleet Control System (FCS))). Because the focus of this thesis is on routing, FCS Fake was kept as simple as possible. Some of the FCS's features where therefore simplified or not implemented at all. For implementation details see Chapter 5.2 (Simulation - FCS Fake).

4.1.1 Shuttles / Agents

Shuttles (Agents) are modeled exactly like their FCS representations. Depending on the shuttle type, different polygonal shapes are used to represent them (see Chapter 3.1.1 (Shuttle)). Similarly, shuttles have the same maximum speed and acceleration values as in FCS. These parameters are used to model the shuttle's motion in the simulation environment. Since edges represent 2D clothoid curves, a time-optimal parametrization method was used to model the

shuttle's motion along them. For further details how shuttle clothoid curve trajectories were modeled, see Chapter 5.6 (Shuttle Motion Model).

Where in FCS shuttles get their orders directly from the OM in FCS Fake, the OM, only provides new orders to the SM if requested. In FCS Fake, agents only communicate with the SM to receive new actions into their action queue. When an action is finished they notify the SM about its completion. Whenever a shuttle comes close to finishing its last action, a new planning round may be triggered centrally for all agents. Figure 27 depicts this process in detail. Whenever a new enqueued action arrives from the ADG framework, it is added to a shuttles action queue. The shuttle then continuously works on its queue until it is empty. At every iteration all consecutive (non-in-place-rotate) move actions are fechted. Since move-actions represent a continuous motion over a clothoid curve, as long as in and out orientations of these clothoid are the same they can be combined. A shuttle trajectory is then estimated over the given curve. After the shuttles finishes performing all involved move actions, they are marked as completed and the ADG is notified. The whole process closely mimics the ADG framework, (see Chapter 2.6.3 (The Action Dependency Graph (ADG) framework)).

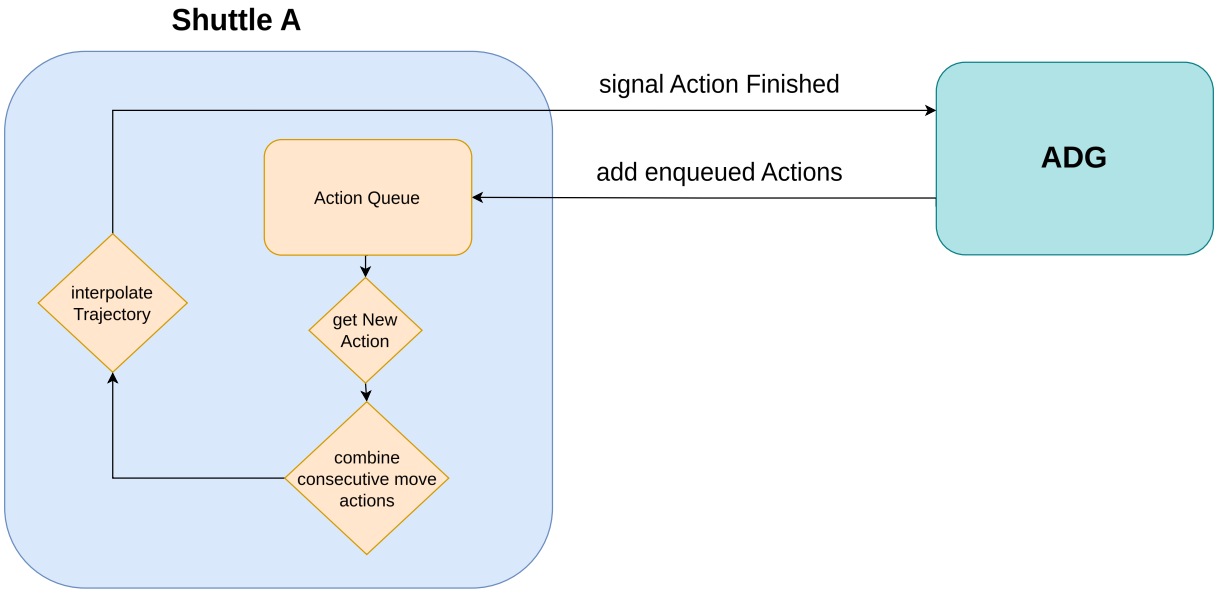


Figure 27: Conceptual diagram of shuttle action processing. The figure illustrates how shuttles add enqueued actions from the ADG to their action queue, process these actions, and notify the ADG upon completion.

It is important to note that edges represent either in-place rotations or clothoid curves (see Chapter 3.3.1 (Routing Graph)), and a distinction is made in how a shuttle interpolates its trajectory. While in-place rotations can only be executed one at a time, multiple clothoid-curve move actions can be combined into a single smooth trajectory. This ensures that shuttles do not have to stop and restart at every node, but can consecutively move at max-speed over

multiple nodes. For further details on how this is implemented see Chapter 5.6.2 (Clothoid Curve Traversal).

4.1.2 Orders and Tasks

As described in Chapter 3.2.1 (Order Hierarchy), an order represent a sequence of load and unload tasks. While performing tasks requires agents to move to the correct node (representing a station), in the context of FCS Fake station interaction is simplified. Instead of a complex interaction with warehouse logistic components such as conveyor belts, a shuttle simply waits at the station for a predefined amount of time.

4.1.3 Order Manager (OM)

Instead of having orders generated from an external system based on real-world warehouse requirements, orders are generated in advance and are fed to the OM at start. While in the FCS the order manager directly sends orders to individual shuttles, in FCS Fake the OM only provides new orders if requested by the SM. In FCS, order distribution is based on various optimization strategies. Since task assignment is outside the scope of this thesis, in FCS Fake, orders are provided either sequentially or randomly.

4.2 MAPD Routing System

The developed MAPD routing system is the central component of this thesis. Figure 28 depicts the high-level architecture of the MAPD routing system. When an agent runs out of actions in the ADG, replanning is triggered. Then a commit cut is created and used as the start positions for the next MAPF planning round. With goal positions representing assigned shuttle task-locations, the mapf-plan is fed to the planner. The planner then provides a single shot mapf-solution Π. This solution is then converted into an ADG and concatenated with the old one. While this happens, agents can continue to work on their actions normally, up to the last commit cut action.

The following sections will go into further detail on each of these components. In Chapter 4.2.1 (Shuttle Manager (SM)), the SM's responsibilities are described in detail. The before mentioned ADG construction and commit cut computation is described in Chapter 4.2.3 (ADG Execution Framework). In Chapter 4.2.2 (MAPF Planner), the MAPF planner architecture and inner workings are discussed in detail.

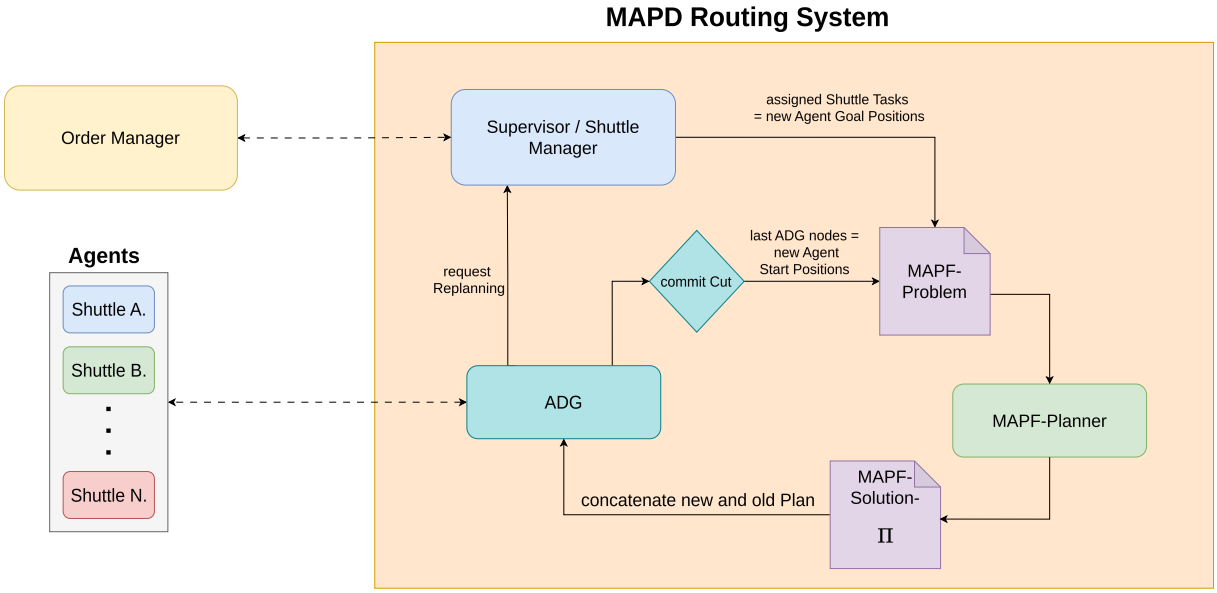


Figure 28: Overview of the MAPD routing system. The figure illustrates the interaction between components, including the Shuttle Manager, ADG, MAPF-Planner, and the process of generating and updating plans.

4.2.1 Shuttle Manager (SM)

The Shuttle Manager (SM) is the central controller of the routing system proposed by this thesis. It is responsible for managing the order, task and planning lifecycle between all involved components and the communication between them.

Figure 29 provides a detailed overview of the SM. Whenever a new order arrives for a shuttle i, its goal position g_i is updated in the current MAPF problem P. As described in Chapter 2.3.1.1 (MAPF - Definition), a MAPF problem is defined as a tuple $\langle G, s, g \rangle$, where G is the planning graph and s, g are start and goal vertices for all agents.

Then this problem P is send to the SM's internal replanning Queue Q_r. Q_r processes all incoming problems in a FIFO manner. For every new problem the current ADG's commit cut c^i is computed. Chapter 4.2.3 (ADG Execution Framework) describes this process in detail. Based on c_i, all shuttle start positions $[s_0 ... s_n]$ are updated in P. After receiving updated start and goal positions, P is send to the MAPF planner (indicated by the thick arrow). This continuously happens as long as there are new problems in the queue. Additionally, replanning can also be triggered if a shuttle completes its action and has less then h actions left in the ADG. Whenever a shuttle finishes its last task in its order, the OM is notified and starts searching for a new order for that shuttle. This process continous until there are no more orders left.

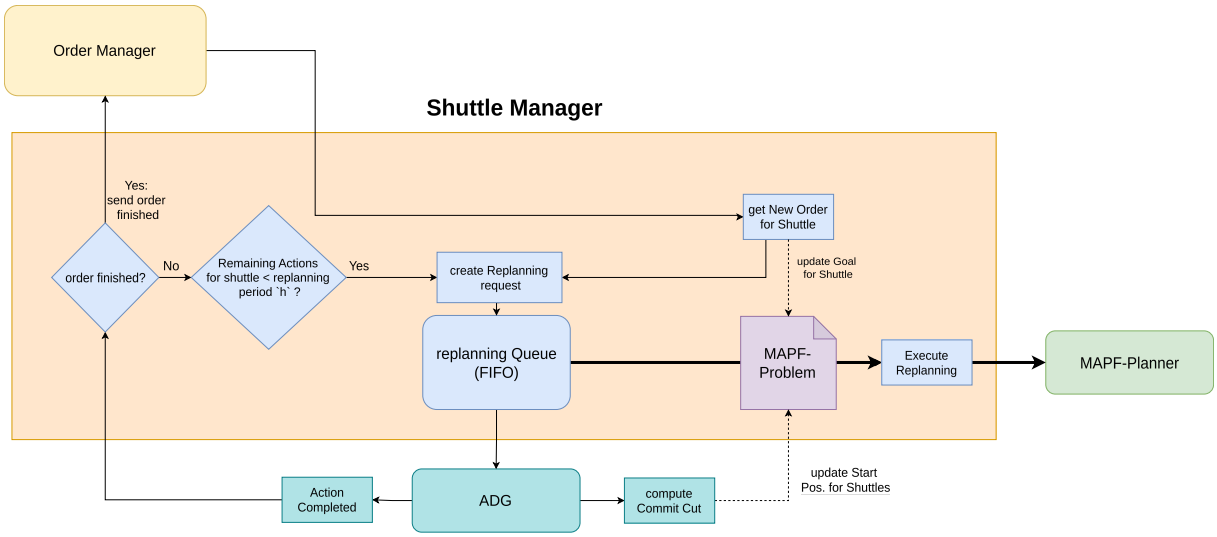


Figure 29: Overview of the Shuttle Manager (SM), illustrating how the SM manages order, task and planning lifecycle and communication between all involved components

4.2.2 MAPF Planner

As described in Chapter 2.3.1.1 (MAPF - Definition), the objective of a planner is to find solutions Π to MAPF-problems $\langle G, s, g \rangle$. In the context of this thesis, G represents a euclidean roadmap graph, with clothoid curve connections and explicitly modeled rotation-edges. While more-complex, routing over this continous graph allows for accurate path planning. The planner developed in this thesis follows this theme. Conceptually the planner works as follows:
It finds collaborative plans using an agent prioritization scheme (see Chapter 2.4.1 (Prioritized Planning (PP))), but incorporates SIPP (Chapter 2.2.4 (Safe Interval Path Planning (SIPP))) for continous time collision avoidance. Since FCS Lanemaps (see Chapter 3.1.2 (Lanemap)) can have arbitrary structures, efficient collision checking is crucial. The idea of pre-computing continous time conflicts, as in PSIPP (Chapter 2.4.3 (Prioritized Safe Interval Path Planning (P-SIPP))), is adopted to speed up the SIPP-single agent path finding process. While conceptually identical, as described in Chapter 3.3.2 (Reservation Grid), FCS uses a geometric reservation grid to determine which shuttle would overlap other shuttles on their path of the Lanemap. In FCS Fake, this information is used to pre compute all possible overlap relationship sets O_R for every entity in the graph on startup. These sets then allow to efficiently determine, which graph entities need their safe-intervals to be reserved. To further mitigate the suboptimal nature of a prioritized-planning approach, two additional strategies have been incorporated:

1. Safe Start Intervals (SSI): As seen in Chapter 2.4.2.2 (Safe-Start Intervals (SSI))), this strategy prohibit the movement of any agent over the start location of another agent for a certain amount of time k. This ensures that agents usually have enough time to move out of the way. In such cases, this allows PSIPP to find a solution where it would not have otherwise.

2. Rolling Horizon Collision Resolution (RHCR): As discussed in Chapter 2.6.6 (Planning for a windowed time horizon (RHCR)), rolling horizon planning is a powerful lifelong MAPF planning strategy. Given a defined planning horizon w, collisions are only checked for any action within w. In the context of prioritized planning, MAPF problems that would otherwise be unsolvable lead to solutions Π where non-blocked agents can continue operation (see example depicted in Chapter 2.6.6 (Planning for a windowed time horizon (RHCR))).

4.2.2.1 MAPF Planner - architecture

Whenever a new MAPF problem is sent to the MAPF-planner, a new sequential planning round is started. Figure 30 depicts this planning process in detail.

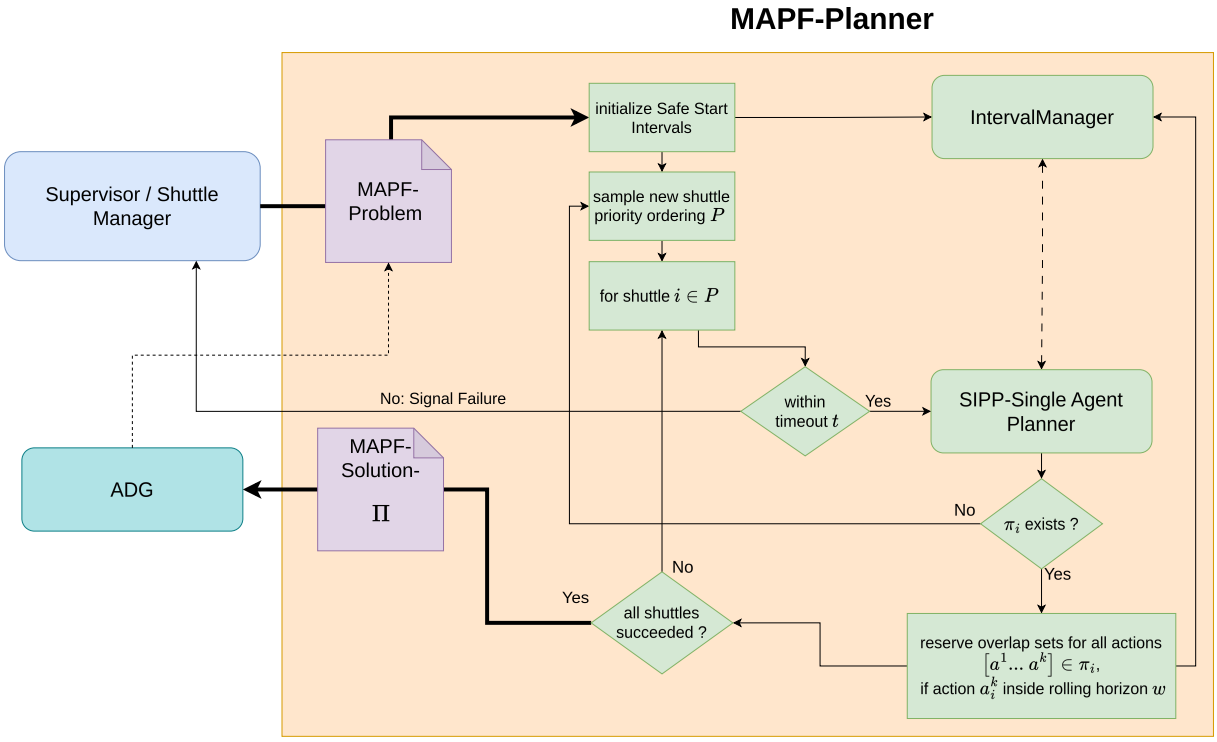


Figure 30: Overview of the MAPF planner, illustrating how it finds sequential MAPF Solutions and its interactions with the other components.

Initially the SM triggers replanning by sending a new MAPF-problem $\langle G, s, g \rangle$ to the MAPF planner. Given $\langle G, s, g \rangle$, a solution Π is computed. Based on all agent start positions s safe-start intervals (SSI) for a user defined interval $[0 - k]$ are allocated in the Interval Manager (IM). ***Note*** the IM is responsible for managing all graph entity safe intervals (see Chapter 2.2.4 (Safe Interval Path Planning (SIPP))). Then a new agent prioritization ordering $P[k \dots 1]$ is drawn randomly (see Chapter 2.4.1.3 (Algorithm)). From the highest priority agent k downwards, SIPP is used to find a path π_i for each agent i. If no path π_i can be found for an agent i, priority ordering P is discarded and a new ordering P' is drawn. When a single agent path is found its collision intervals are reserved in the IM up to a predefined planning horizon w.

This process is repeated until a valid solution Π for all agents is found or a maximum timeout τ is reached.

4.2.2.2 MAPF Planner - Algorithm

Algorithm 6 depicts the used planner in detail.

Algorithm 6: Mapf Planner (FCS)

input: $\langle G, s, g \rangle$, max timeout τ,
safe-interval duration k
rolling horizon w
overlap-sets O_R
output: $\Pi : [\pi_1, \dots, \pi_n] \vee \emptyset$

while True:
  $\Pi \leftarrow \emptyset$
  success $\leftarrow$ True
  reset all safe-intervals in IM

  */ add safe start intervals k */
  for each start pos $s_i \in s$:
    $a_{w(s_i)} \leftarrow$ wait at start action for node s and duration k
    for each graph entity $e \in O_R\left(a_{w(s_i)}\right)$:
      Add collision iv. $[0 - k]$ to IM(e)

  */ perform prioritized planning */
  priority orderings $P\ [k..1] \leftarrow$ *assignAgentPriorities()*
  for each agent i in P:
    reset all entities e from IM, that are unique to agent i
    $\pi_i \leftarrow$ *SIPP-FCS(G, s_i, g_i, IM)*
    if $\pi_i = \emptyset \vee$ reached timeout τ
      success $\leftarrow$ False
      break
    if reached timeout τ
      return $\emptyset$

    */ allocate collisions for actions within planning horizon w */
    for each action a_i^j in π_i:
      if a_i^j.start $< w$
        for each graph entity $e \in O_R\left(a_i^j\right)$:
          Add collision iv for a_i^j to IM(e)
        else
          break
    $\Pi_{[i]} \leftarrow \pi_i$

  */ validate solution Π */
  agents_made_progress $\leftarrow$ False
  for each plan $\pi_i[a_i^0 \dots a_i^n]$ in Π:

```
27        if a_i^n.node ≠ start pos s_i
28            agents_made_progress ← True
29            break
30    if success ∧ agents_made_progress
31        return Π
32    else
33        return ∅
```

The input to this planner is a MAPF problem $\langle G, s, g \rangle$ with a maximum timeout τ, a safe-interval duration k, a rolling horizon w and overlap sets O_R.

Whenever a new prioritized planning round starts, previously used components have to be reset. In Line (4), the IM has all its safe intervals i_e reset to a default state. To be more specific: The IM is a mapping $\text{IM}(e) : m_e$ from any entity $e \in G$ to a set of safe-intervals m_e. As described in Chapter 2.2.4 (Safe Interval Path Planning (SIPP)), initially m_e only contains a single interval $i_0 : [0, \infty]$. When e.g. a collision interval $c_i : [2.0, 4.0]$ is added then i_0 will be split into two intervals $i_0 : [0, 2.0]$ and $i_1 : [4.0, \infty]$. Thus, resetting the IM means that all intervals are set back to $i_0 : [0, \infty]$.

In Line (5 - 9), for all overlapping entities of all shuttle start positions $O_{R(s_i)}$, collision intervals $[0, k]$ are added into the IM. Therefore when the first agent plans its path, it has to avoid other shuttles start locations for the first k time steps.

In (11 - 27) a standard prioritized planning approach similar to Algorithm 3 is performed. Its differences are as follows: Since the usage of Safe Start Intervals (SSI) required adding collision intervals to the overlapping entities of the start positions of all agents (5 - 8), if an agent i then starts its planning, added collision intervals $[0, k]$ of entities e that are unique to i (not overlapped by other shuttles) are removed from IM (11). Otherwise an agent would find its start location blocked by its own safe start intervals. A modified version of SIPP is used (12). When a single agent plan π_i is found, only collision intervals for actions within the planning horizon w are allocated (19). This ensures that subsequent agents will not find their paths blocked by higher-priority agents, beyond w. ***Note*** that combining SSI and RHCR requires $k \ll w$. Otherwise planning cannot work properly. ***Note*** also, collision interval allocation for each single agent plan π_i does not only track collisions for directly involved nodes and edges in the path but again allocates every involved entity e in any of the actions a_i^j overlap sets $O_R\left(a_i^j\right)$ (20 - 21).

In (25 - 33) the solution is validated. Since a rolling horizon approach is used (19), a planner will succeed even if all single agent plans are just agents waiting at their own start position until the end of the planning horizon w. In this case no agent made progress, and the solution is therefore discarded.

As already discussed, the PP-like planners is a two level algorithm. The higher level is the priority based planning and retrying of different prioritization orderings. The lower level is single agent path finding given the current state of all free interval in the IM. This planner is referred to as Safe Interval Path Planning - FCS (SIPP-FCS) because it leverages an adapted version of SIPP. As already discussed in Chapter 2.2.4 (Safe Interval Path Planning (SIPP)), SIPP is an A*-like search algorithm. A state s in SIPP consists of the following properties:

- Current node c: The node the agent is at.
- Previous node p: The node the agent was in the previous state.
- Interval I_t: The interval [start, end] during which the agent is in this state. (e.g. Move, wait.)

SIPP-FCS directly follows the original SIPP architecture (as depicted in Algorithm 1) with the following differences:

1. Repetitive orientation edge successor states are omitted.

Recall that the graph G used in FCS represents an orientation graph where rotational edges are modeled explicitly (see Chapter 3.3.1 (Routing Graph)). SIPP will generate a successor state per edge it can find for every node. This orientation graph formulation allows SIPP to find paths for larger agent's where different orientations may overlap different entities. The downside to this is that SIPP search expansion can explode because it can iterate back and forth between two orientational edges of the same node. To avoid this, any in-place rotation successor states are omitted if the previous state was also a rotational edge. This ensures that the search space is not unnecessarily expanded.

2. Since an orientation graph is used, successor state generation works differently.

This is because moving from node c to a neighboring node n' does not only depend on the safe intervals $m_{n'}$ and m_c but also on $m_{c \to n'}$. Algorithm 7 describes the successor state generation process in detail: Line (8-10) depicts the difference in move successor state generation. A valid move successor state only exists on intersecting free intervals between $m_{c \to n'}$ and $m_{n'}$ because edges are arbitrarily long and may geometrically overlap with any amount of other entities. Or in other words, moving to a neighboring node is only possible if itself and the road towards it is free.

3. Travel durations are estimated using Time-Optimal Path Parameterization (TOPPRA).

In real life travel durations between nodes are not constant and depend on kinodynamic constraints of agents. In an orientation graph, edges represent either orientational edges or clothoid curves. Since edges are modeled so close to the actual motion of agents, accurately estimating travel durations is possible. Given an agents motion model (see Chapter 5.6 (Shuttle Motion Model)), a TOPPRA based approach is used to estimate how long an agent may take to travel over that edge.

- If an edge represents a clothoid curve, its travel duration is evaluated as one continous path over all its control points. For more details see (Chapter 5.6.1 (Trajectory optimization))
- If an edge $(c \rightarrow n')$ represents an in place rotation, it works as follows: First, the smallest angular difference $\Delta\theta$ between orientations θ_c and $\theta_{n'}$ is calculated. Then, an agents motion model is applied to this orientational transition such that start and end velocities are both 0.0. This process is done in line (4) ***estimate_travel_duration***. During operation this travel durations are cached to speed up the planning process.

Algorithm 7: getSuccessors - (SIPP-FCS)

input: current state $s : [c, p, I_t]$, roadmap graph G, interval manager IM
output: successors $S' : [s_1, \dots, s_n]$,

1 $S' \leftarrow \emptyset$
2 $i_c \leftarrow$ current free interval for node c in IM (c) starting at $s.I_t$.end
3 **for each** neighboring node n' of $(s.c)$:
4 $t_{\text{travel}} \leftarrow$ *estimate_travel_duration*$(c \rightarrow n')$
5 move interval $k_{c \rightarrow n'} \leftarrow [s.I_t.\text{end}, s.I_t.\text{end} + \text{t}_{\text{travel}}]$
6 **if** $k_{c \rightarrow n'} \not\subseteq i_c$
7 **break**
8 $m_{n'} \leftarrow$ set of all free intervals of node n' in IM (n')
9 $m_{c \rightarrow n'} \leftarrow$ set of all free intervals for edge $(c \rightarrow n')$ in IM ((c->n'))
10 $M' \leftarrow$ *find_intersecting_intervals* $(m_{n'}, m_{c \rightarrow n'})$ */ linear sweep */
11 **for each** safe interval i' in M':
*/ check if moving to neighbor n' is possible */
12 **if** $k_{c \rightarrow n'} \subseteq i'$:
13 $s' \leftarrow [n', c, k_{c \rightarrow n'}]$
14 insert s' into S'
15 **continue**
*/ check if waiting for neighbor n' at c is possible */
16 **if** $[i'.\text{start}, i'.\text{start} + t_{\text{travel}}] \subseteq i_c$
17 wait interval $k_{c \rightarrow c} \leftarrow [s.I_t.\text{end}, s.I_t.\text{end} + i'.\text{start} - s.I_t.\text{end}]$
18 $s' \leftarrow [c, c, k_{c \rightarrow c}]$
19 insert s' into S'
20 return S'

4.2.2.3 MAPF Planner - Example

To further understand how planning on an orientational graph for large agents works, consider the situation depicted in Figure 31. In (a) there are two shuttles S_b (blue) and S_g (green) that want to find their paths over the Lanemap. In (b)the underlying routing graph is depicted. Red dots show corresponding orientation edges. The Lanemap goal D has two routing graph edges $D_{0.0}$ and $D_{90.0}$ since an agent may traverse D in these two orientations. Black arrows

indicate clothoid curve edges and yellow arrows in-place rotations (goals B and D are turnable). Every edge has costs (estimated travel durations) associated with it.

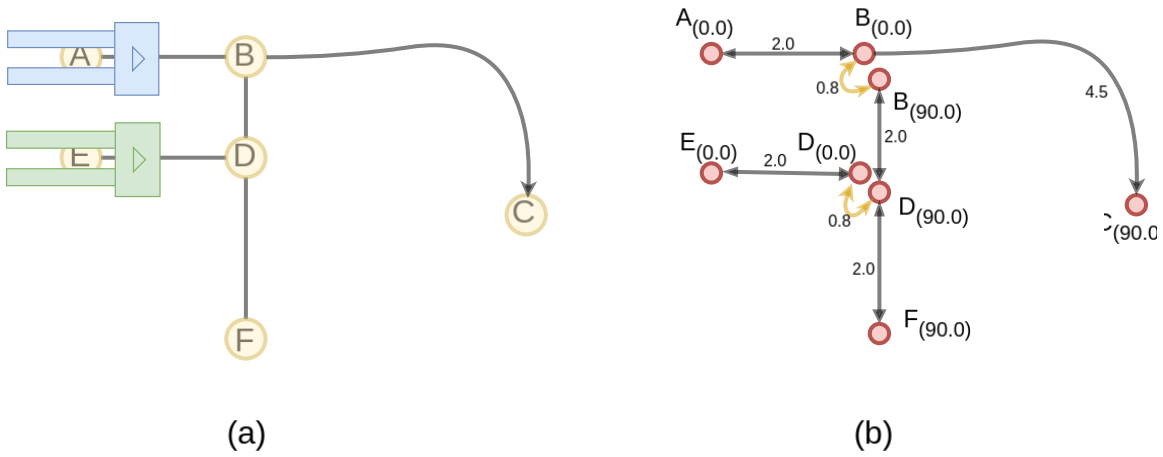


Figure 31: The same MAPF problem from the lanemap and the orientation graph perspective. (a): The lanemap with two agents S_b (blue) and S_g (green). S_b wants to move from node A to C, while S_g wants to move from E to F. (b): The underlying orientation graph used for planning their paths.

The depicted MAPF problem consists of the orientational graph G (depicted on the right) and the agents (S_b, S_g) start and goal positions s and g : $\langle G, [A_{0.0}, E_{0.0}], [C_{90.0}, F_{90.0}]\rangle$.
Initially the planner randomly samples a priority ordering P that determines the order in which single agent paths are found. e.g. in this case $P = [S_b, S_g]$. Then SIPP-FCS is used to find a path for S_b. Given that (safe start intervals aside) all intervals m_e are free initially, S_b travels along the shortest path π_b to its goal.

$$\pi_b = (\{A_{0.0}, B_{0.0}, [0.0, 2.0]\}, \{B_{0.0}, C_{90.0}, [2.0, 6.5]\}, \{C_{90.0}, C_{90.0}, [6.5, \infty]\})$$

Then the path for S_b is reserved in the IM. To determine which graph entities e can not be visited by subsequent shuttles precomputed overlap sets O_R are used.
The action $a_0^b \in \pi_b$ represents moving from $A_{0.0}$ to $B_{0.0}$. Therefore, not all entities $O_{R((A_{0.0} \to B_{0.0}))}$ can be visited during interval $[0.0, 2.0]$. In this example $O_{R((A_{0.0} \to B_{0.0}))}$ contains $\{A_{0.0}, B_{0.0}, B_{90.0}, (B_{90.0} \to D_{90.0}), D_{90.0}, (D_{0.0} \to D_{90.0}\}$.
Note that $(B_{90.0} \to D_{90.0}) \wedge D_{90.0} \subset O_{R((A_{0.0} \to B_{0.0}))}$ because a shuttle staying at node D with orientation $90.0°$ would collide with another agent moving from A to B. Similarly a_1^b blocks all entities $e \in O_{R((B_{0.0} \to C_{90.0}))}$ After applying both collision intervals $[0.0, 2.0]$ and $[2.0, 6.5]$ the safe interval set $m_{D_{90.0}}$ contains a single interval. $[6.5, \infty]$. When S_g then plans its path, it has to wait at $D_{0.0}$ until $t = 6.5$ where there is a safe interval in $m_{D_{90.0}}$. Therefore the found path for S_g is:

$$\pi_g = (\{E_{0.0}, D_{0.0}[0.0, 2.0]\}, \{D_{0.0}, D_{0.0}[2.0, 6.5]\}, \{D_{0.0}, D_{90.0}, [6.5, 7.3]\},$$
$$\{D_{90.0}, F_{90.0}, [7.3, 9.3]\}, \{F_{90.0}, F_{90.0}[9.3, \infty]\})$$

4.2.3 ADG Execution Framework

Chapter 4.2.3.1 (Integration into the MAPD Routing System) will provide an overview of how the ADG framework is incorporated into the overall architecture. Chapter 4.2.3.2 (FCS ADG - Enhancing the ADG Framework for the FCS roadmap setting) discusses changes made to the original implementation of the ADG to incorporate continous time planning results (as returned from the used MAPF planner) with the orientation graph based reservation framework present in FCS (see Chapter 3.3.1 (Routing Graph)).

4.2.3.1 Integration into the MAPD Routing System

As described in detail in Chapter 2.6.3 (The Action Dependency Graph (ADG) framework) and Chapter 2.6.7 (The RHCRE framework), using the ADG framework ensure that robust plan execution as well as the overlap of planning and execution of shuttles. The ADG framework developed in this thesis works exactly as discussed in Chapter 2.6.3 (The Action Dependency Graph (ADG) framework) with a few minor changes, discussed in the next chapter.

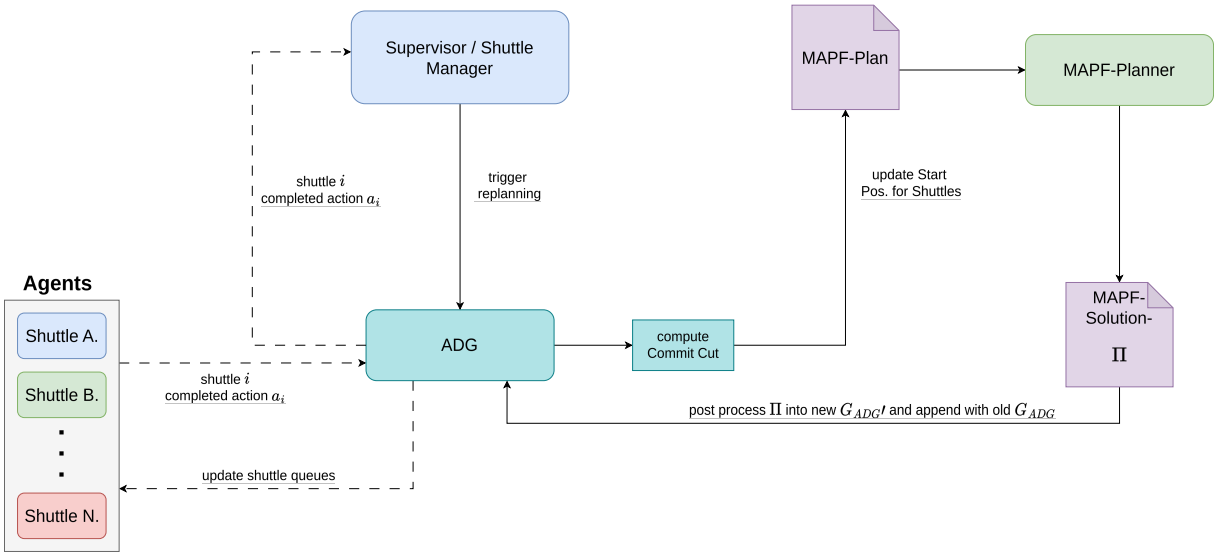


Figure 32: Overview of the ADG framework, illustrating how it manages all communication with shuttles and its involvement in the sequential MAPF planning process.

Figure 32 provides an overview of how the ADG framework is incorporated into the overall architecture. Whenever the Shuttle Manager (SM) triggers replanning, a ***commit cut*** c_i is computed from the current ADG (G_{ADG}). (see Chapter 2.6.5 (Lifelong planning with the ADG framework)). As c_i represents the latest nodes from the old plan that are consistently reachable for all agents without any conflicts, c_i is used as the set of start nodes s_i for all agents in the next MAPF problem P'. When the MAPF planner finds a solution Π', it is constructed

into a new ADG $G_{ADG'}$ (see Chapter 2.6.4 (Construction of the ADG)) and concatenated with the old one. During this process agents can continue executing all their actions in the old plan up to the last commit cut action.

When on the other hand a shuttle i notifies the ADG framework that it has completed an action a_i, the ADG framework recursively checks all direct dependencies of a_i and sets them to the state *enqueued* if possible. This process is called **enqueue_vertices()** and ensures that agents can continue executing their actions without much communication overhead. Additionally, this action-success is also communicated to the SM. The SM then further decides if a replanning is necessary. (see Chapter 4.2.1 (Shuttle Manager (SM)))

4.2.3.2 FCS ADG - Enhancing the ADG Framework for the FCS roadmap setting

As discussed in Chapter 2.6.3 (The Action Dependency Graph (ADG) framework), the original ADG framework is meant for classical MAPF problems. There, time is considered discrete and a roadmap-graph G represents a lattice like grid where all vertices (nodes) are independent of each other. This means that an agent cannot occupy more then one node at a time. In the context of FCS, agents travel over a shared clothoid curve roadmap (Chapter 3.1.2 (Lanemap)) in continous space. Depending on the shuttle footprint (Chapter 3.3.2 (Reservation Grid)), a shuttle staying at a node or traveling over an edge will therefore block or overlap multiple other nodes or edges (entities) on its path. In order to integrate the ADG framework into the FCS routing system, two changes were necessary:

1. **Continous time dependency checking**

When an ADG is constructed from a MAPF solution, all same shuttle actions are connected to each other first (*Type 1 Dependencies*). Then, all cross-shuttle actions are exhaustively compared to each other and a dependency (*Type 2*) from action a_x to a_y is created if a_x.start $=$ a_y.goal and a_x.time $\leq a_y$.time. This ensures that an agent can only execute action a_y if a_x has been finished by another agent see Chapter 2.6.4 (Construction of the ADG).

In the FCS, the underlying planning graph is euclidean (see Chapter 2.3.2.1 (MAPF on Euclidean Graphs.)). This means that nodes can be at any position in 2D-space. There, a discrete time planning formulation can lead to suboptimal paths (see Chapter 2.3.2.4 (Continous time)). Because the MAPF planner used in this thesis (Chapter 4.2.2 (MAPF Planner)) uses a continous time formulation, the cross-shuttle (*Type 2*) dependency check needs to account for floating point inaccuracies. More specifically, for two actions a_x, a_y, their start times t_x, t_y are compared as depicted in Equation 1:

$$t_y - \varepsilon < t_x < t_y + \varepsilon \Rightarrow t_x = t_y \tag{1}$$

Note that ε is a user defined value chosen small enough such that it is always less then any action duration performable in the graph. Otherwise the above mentioned dependency check could result in a cyclic dependency in the ADG.

Equation 1 describes if action start_times $\{t_A, t_B\}$ fit for a *type2* edge $(a_x \rightarrow a_y)$ creation. This condition will be referred to as: **timing_fits(a,b)**

2. **Adding overlap relationships to the dependency search**

As described above, a euclidean roadmap graph is considered, where agents can overlap multiple nodes or edges at the same time. Thus for an action vertex p^i and p^k, a *type2* edge ($p^i \rightarrow p^k$) will not only be created if p^i and p^k share the same exact node but rather if one shuttle S_i would geometrically collide with another shuttle S_k while S_i performed action a_n^i associated with p^i while S_k performs action a_n^k associated with p^k. As discussed in Chapter 3.3.2 (Reservation Grid) this geometric collision information is precomputed and stored as overlap relationship sets O_R. Thus, the overlap sets for p^i's entities $O_R(p^i)$ contain all other entities that p^i would collide with if an agent would execute action associated with p^i while another one stays at a node or moves over an edge in $O_R(p^i)$. In the context of FCS, edges represent in-place rotations or clothoid curves in 2D-space (see Chapter 3.3.1 (Routing Graph)). This increases the complexity of collision checking, as discussed in Chapter 4.2.2 (MAPF Planner). As described there, other entities are blocked for the whole duration of the edge travel. This is a necessary simplification as it makes move (edge-traversal) and wait (staying at a node) actions well defined regarding collision checking and ADG dependency creation (type2-edge creation).

During dependency search move and wait actions can be treated equally, by defining any wait action a_x as an equivalent move action a_y on edge (a_x.node $\rightarrow a_x$.node) over a_x's wait duration. Dependency search over an action vertex p_A is therefore done as follows: If during executing a_A an agent moves over an edge $(x \rightarrow y)$, it will consider any action vertex p_B with action a_B $(x' \rightarrow y')$ as a dependency candidate, if action a_B is in any of the overlap sets of node x, node y or edge (x, y), or in other words, if $a_B \in O_R(a_A)$ holds. More specifically, if action a_B is within the overlap sets O_R of action a_A is defined by Equation 2.

$$a_B \in O_R(a_A) \iff \{x', y', (x', y')\} \cap \{O_R(a) \cup O_R(b) \cup O_R((a, b))\} \neq \emptyset \tag{2}$$

Since overlap sets are precomputed based on geometric collision checking, overlap checks are symmetrical.

$$a_B \in O_R(a_A) \iff a_A \in O_R(a_B) \tag{3}$$

Equation 3 describes if any two actions (a_B, a_A) have overlap relationships. This condition will be further refer to as **overlaps(a,b)**.

4.2.3.3 FCS ADG Algorithms

As already described in the previous section, the ADG construction algorithm works like the original one (see Algorithm 5) except for the following changes:

1. wait-actions are omitted.
   This omission is possible because agents wait behavior is already implicitly defined in the ADGs dependency structure. Additionally, omitting explicit wait-actions can prevent delays in operational progress when other agents have already completed their actions ahead of time.
2. *Type 2* dependencies are created differently.
   Since this solution is based on an orientational graph structure, overlap sets O_R are considered, instead of only comparing start and goal vertices to find dependency candidates.

Algorithm 8 depicts the *type 2* dependency creation for the FCS orientation based graph system, as discussed in the two previous chapters.

Note the following changes in the nomenclature w.r.t. to the original algorithm for clarification purposes: Nested agent iteration loops found in the original algorithm are omitted for clarity reasons. Instead, the algorithm just iterates over all already created action vertices $p^i \in G_{\text{ADG}}\ \{V, E\}$. Every action vertex p^i contains the following properties:

- **agent** s: which agent it belongs to
- **start time** t: when the action starts according to the mapf-plan π_i it originated from.
- **action** a: The action $a^i_{\{1\dots N\}}$ itself.

These properties will further be directly referred to in the pseudo code.

Algorithm 8: (FCS) ADG Type 2 Dependency Creation

input: V list of all action vertices p^i with (only type-1) edges E
output: $G_{\text{ADG}}\ \{V, E\}$

for $p^i \in V$:
  for $p^k \in V$:
    if $p^i.s = p^k.s$
      continue */ do not compare same agent vertices. /*
    if timing_fits$(p^i.a, p^k.a)$ and **overlaps** $(p^i.a, p^k.a)$
      Add edge $(p^i \rightarrow p^k)$ to E
return G_{ADG}

4.2.3.4 FCS ADG Example

To further understand how the adapted ADG construction algorithm works, consider the example depicted in Figure 31. As already described there, a possible solution $\Pi\left[\pi_g, \pi_b\right]$ for both shuttles is:

$$\pi_g = (\{E_{0.0}, D_{0.0}[0.0, 2.0]\}, \{D_{0.0}, D_{0.0}[2.0, 6.5]\}, \{D_{0.0}, D_{90.0}, [6.5, 7.3]\},$$
$$\{D_{90.0}, F_{90.0}, [7.3, 9.3]\}, \{F_{90.0}, F_{90.0}[9.3, \infty]\})$$
$$\pi_b = (\{A_{0.0}, B_{0.0}, [0.0, 2.0]\}, \{B_{0.0}, C_{90.0}, [2.0, 6.5]\}, \{C_{90.0}, C_{90.0}, [6.5, \infty]\})$$

Figure 33 shows the result G_{ADG} of the adapted ADG-construction algorithm for MAPF solution Π. Both created *type 2* edges $(b_0 \to g_1)$ and $(b_1 \to g_1)$ only exist because of the overlap relationships between their actions. As already discussed wait actions are omitted from the ADG construction process. S_g can start rotating at D as soon as S_b arrives at C, which might be earlier then predicted by the MAPF plan.

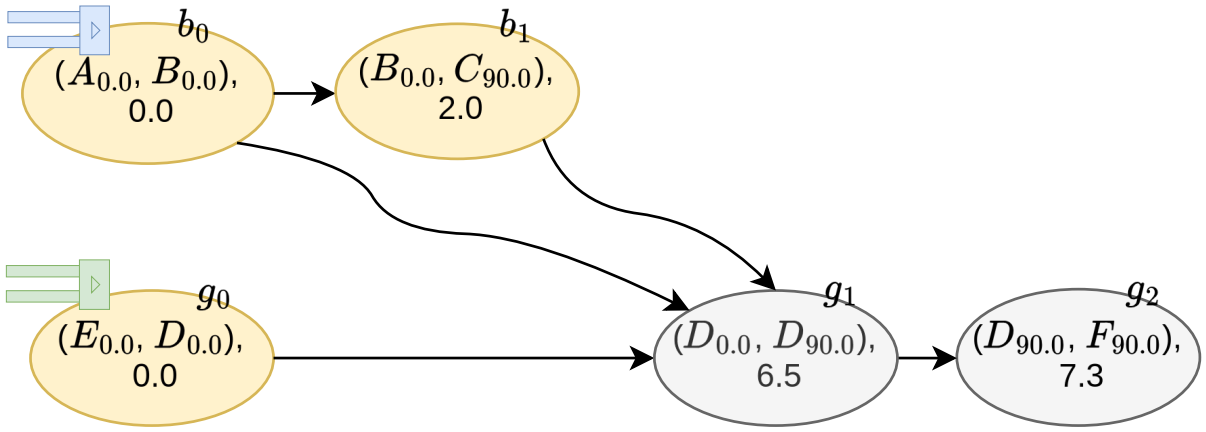


Figure 33: Example of how ADG construction works in FCS. The blue shuttle S_b can directly move to its goal. S_g (green) can only start rotating at D when S_b has reached C.

5 Implementation

This chapter provides a detailed account of the technologies and methods underlying the proposed routing system. It begins with a discussion of the development environment used during the implementation process in Chapter 5.1 (Development Environment), followed by an in-depth look at the construction of the FCS Fake simulation in Chapter 5.2 (Simulation - FCS Fake). The setup and communication with the original FCS system are then explored in Chapter 5.3 (System Initialization). Chapter 5.4 (Graph) details how the clothoid roadmap graph was created and represented. Additionally, Chapter 5.5 (Overlap Relationship Sets) covers the process of extracting overlap sets used in collision avoidance. Finally, Chapter 5.6 (Shuttle Motion Model) discusses the technology used to model shuttle motion over the clothoid roadmap graph.

5.1 Development Environment

The development process for this thesis can be divided into two main parts:

1. The communication layer with the original FCS system.
   As the whole FCS software stack is written in C++, integral components where developed in C++ using the CLion IDE. This mostly involved the MAPD routing client responsible for setting up the FCS Fake, see: Chapter 5.3 (System Initialization).
2. The FCS Fake simulation.
   To allow rapid prototyping, testing and generally a fast incremental development process, Python was chosen as the main programming language throughout the FCS Fake system. Pythons fast development cycle and rich ecosystem of libraries allowed for quick implementation of the simulation environment. PyCharm was chosen as the main IDE for the Pythons development.

5.2 Simulation - FCS Fake

As already described in Chapter 4.1 (Simulation (FCS Fake)), the bare-boned simulation works as follows:

Given a set of orders, the FCS Fake, distributes orders between shuttles, ensuring robust and collision-free routing over the shared road network. Shuttles perform tasks on defined goal nodes associated with orders by simply occupying the node for a predefined amount of time. Simulation continues until all orders are done. This of course, is a simplification. For the sake of this thesis, a lot of features not directly relevant to the routing problem were simplified or not implemented in FCS Fake.

The most noteworthy differences are:

- A Discrete Event Simulation (DES) is used instead of a natural time progression.

- Visualization is done with post-simulation.
- No connection to external systems like conveyor belt interaction or similar exists.
- No complex order distribution strategies are used. Orders are either provided in sequence or randomly.
- Instead od the lanemap level, FCS Fake directly works on the routing graph level. (see Chapter 5.4 (Graph))

5.2.1 Discrete Event Simulation - DES

The biggest (implementation) difference to the original FCS system was the decision to build FCS Fake as a DES instead of implicitly modeling time progressional naturally. While being initially more complicated, this allowed for a much faster testing and development cycle.

In a conventional system, time is implicit meaning that time progresses naturally as the system executes code and multiple threads or processes communicate and wait for each other. A shuttle, for example, moves from one node to another by applying a velocity based motion model. Modeled correctly, the shuttle will reach its destination after the correct amount of time elapsed. This of course works, but in the context of a routing system most of the simulation time will be spent waiting for shuttles to reach their goals. In a DES on the other hand, time is modeled explicitly through the execution order of discrete events. Instead of building a multi-threaded application, a DES runs a single thread that processes events in the order they arrive in a priority queue. This not only makes debugging easier, but also allows simulation results to be calculated much faster. In a DES, every component that would naturally be built as an independent thread, is instead modeled as an independent event-process that contains its own time progression. In this context, FCS Fake maintains the following independent event-processes:

1. One process per shuttle.
2. One process for the Order Manager (OM).
3. One process for the Shuttle Manager (SM).

Internal time for these components only progresses if they have finished an event or are notified by another component. A shuttle maintains its own time. If it starts moving over an edge, it applies its motion model over the full path to estimate the time it will take to reach the end. It then updates its internal time with this value. It then notifies the SM that it has finished its action. The SM, then updates its internal time based on when it received the callback. Not only can fixed time intervals be used to progress a DES, natural time progression based on the computation time of individual components is also possible. When the SM triggers a new planning round, the system tracks how long the calculation took and progress the simulation time accordingly. To build this DES, the Simpy library (Zinoviev, 2024) was used.

The performance of a centralized routing system is significantly impacted by the duration of each planning round. By employing a DES, computationally intensive components were implemented directly in Python. Since time progression is explicitly updated based on the planner's tracked computation time, the computation time τ can be manually reduced by a user-defined factor κ. This manual reduction allowed for the evaluation of how differently fast planners impact the overall system's performance.

5.2.2 Visualization

While a DES offers many advantages for the development process, it does not natively support simulation visualization. To address this, all interpolated shuttle paths and any occurring events are logged during simulation. This log file can be loaded in a post-simulation to visualize the entire run. The visualization was implemented using the Pygame library (Pygame-Community, 2024). Figure 34 shows the developed pygame visualization in detail: Multiple shuttles (each has its own color) travel over the shared lanemap (blue dots with black edges). They are rendered as a 2D - polygon with the size of their footprint.

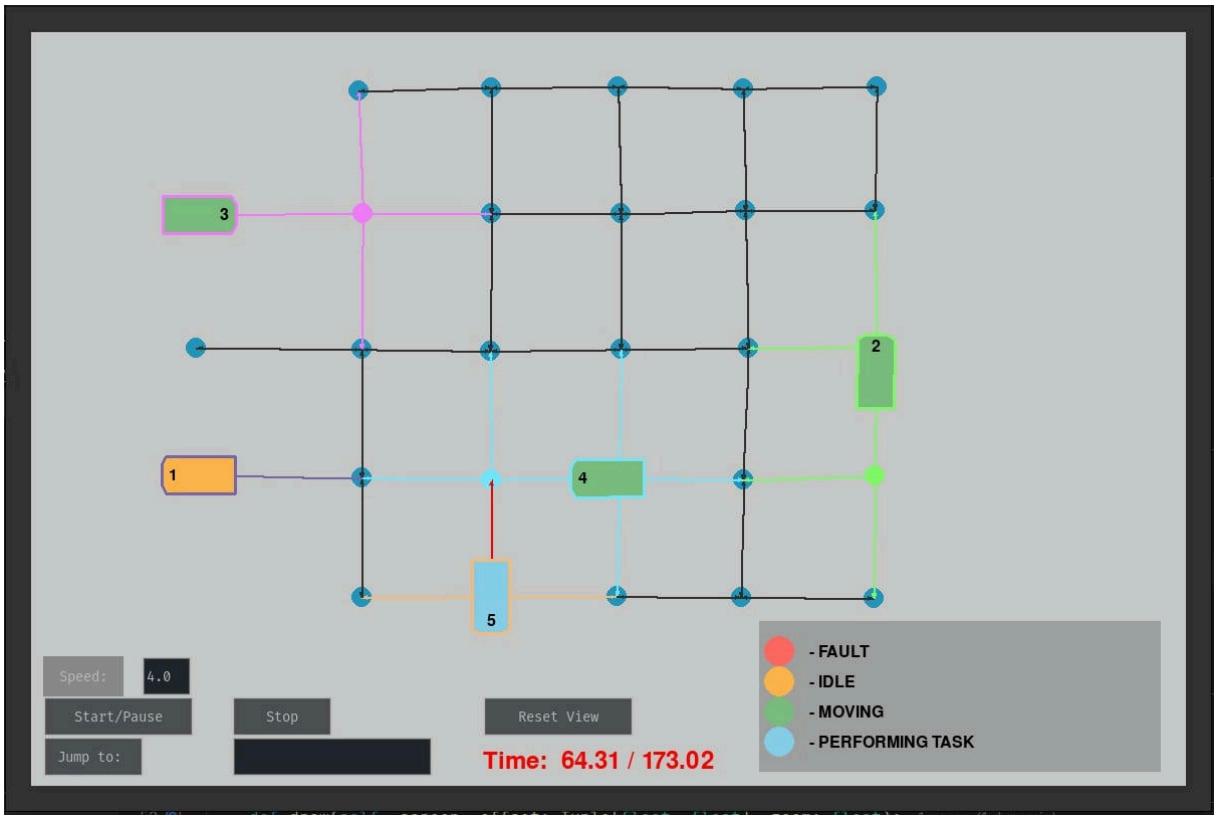


Figure 34: The pygame visualization used for rendering the FCS Fake simulation results.

Shuttles can be in the following state:

- IDLE (yellow) - they are waiting for a new action from the ADG.
- MOVING (green) - they are performing an action.
- PERFORMING TASK (blue) - occupying a node to simulate a task execution.

- FAULT (red) - used to randomly simulate a shuttle failure.

The DES simulation result is precomputed and loaded from a file. This allows running the visualization with any speed-up. Further it allows to pause, rewind or stop the simulation at any time, allowing fine-grained visual inspection of results. The depicted UI-components can be used to control this.

5.3 System Initialization

Before running FCS Fake, the routing graph and all overlap set information O_R is exported from the original FCS system. The roadmap depicted in Figure 34 was first drawn in the FCS and then exported to be processed in FCS Fake via a C++ Protobuf service and a Python client. For every lanemap instance the client is run one time and stores the routing-graph level from FCS into a *json* config file. When FCS Fake is started this information is simply loaded.

5.4 Graph

To represent the original routing graph (see Chapter 3.3.1 (Routing Graph)) exported from FCS in Python, the library NetworkX was used. Efficient neighborhood lookup is essential during routing, and NetworkX addresses this by providing optimized methods for quickly accessing neighboring nodes and edges.

In contrast to the original FCS where a 2 level graph structure is used (Lanemap -> orientation graph), FCS Fake directly works on the routing graph level. To ensure associated clothoid curve information is maintained, additional node and edge properties are necessary.

Nodes have the following properties:

- *pos*: The 2D position of the node.

Edges have the following properties:

- *id_from*: The edges source node id.
- *id_to*: The edges target node id.
- *angle_from*: The edges initial orientation, starting from its source node
- *angle_to*: The edges final orientation, at its target node.

Note: Edge travel costs are not stored directly on edges. They are rather calculated on the fly and cached for efficient lookup, see Chapter 4.2.2.2 (MAPF Planner - Algorithm). Pre-computing them would also be possible. Theoretically this is a trade-off between runtime and startup delay.

To find out if an edge e represents a clothoid curve or a rotational edge, is done by checking if the source *id_from* and target *id_to* nodes of e have the same *pos*.If this is true, it is a rotational edge, see Chapter 3.3.1 (Routing Graph). Otherwise, it is a clothoid curve edge.

5.5 Overlap Relationship Sets

Overlap sets O_R are crucial for preventing conflicts between shuttles in the FCS routing system. These sets are used to quickly identify and resolve potential collisions by ensuring that no two shuttles occupy overlapping spaces at the same time. This section describes how overlap sets O_R were extracted for every entity in the FCS routing graph.

From a given FCS instance all Lanemap, graph and reservation grid information is exported to FCS Fake via a Protobuf service-client. Then to extract overlap sets O_R for every entity e (node or edge) in G, the FCS reservation grid (Chapter 3.3.2 (Reservation Grid)) is queried for the geometric space g_e a shuttle i would overlap would it perform action a_e^i. This is compared with every other entity e' that is within a predefined radius from e. If $g_{e'}$ and g_e overlap then e' is put into $O_R(e)$ and vice versa.

5.6 Shuttle Motion Model

In FCS the routing graph used for path planning represents a euclidean clothoid curve roadmap (Chapter 3.3.1 (Routing Graph)) where in-place rotation is modeled explicitly. This setting has multiple advantages:

- During simulation accurate approximation of real life shuttle trajectories becomes possible, see Chapter 4.1.1 (Shuttles / Agents).
- During path planning accurate travel durations can be used to guide the search, see Chapter 4.2.2.2 (MAPF Planner - Algorithm).

A shuttles motion model in the FCS simulation is described by the following parameters:

- Max. linear velocity $v_{\max}$
- Max. angular velocity $w_{\max}$
- Max. linear acceleration $a_{\max}$
- Max. angular acceleration $\alpha_{\max}$

5.6.1 Trajectory optimization

Calculating the trajectory of a shuttle i over a clothoid curve C works as follows:

1. C is sampled into waypoints $w_0 ... w_n$ such that there is approx. one waypoint every centimeter of the curve.
2. Given a shuttles motion constraints $\begin{pmatrix} v_{\max} \\ w_{\max} \end{pmatrix} \begin{pmatrix} a_{\max} \\ \alpha_{\max} \end{pmatrix}$ and start and end velocity $\begin{pmatrix} v_{\text{start}} \\ v_{\text{end}} \end{pmatrix}$, Time-Optimal Path Parameterization (TOPPRA) is used to calculate the shuttles trajectory T.

This information can be directly used to estimate at which point on C a shuttle i is at a given time t. For more details on TOPPRA, the reader is referred to (Pham, 2014)

5.6.2 Clothoid Curve Traversal

Using the ADG framework, a shuttle may be able to drive over multiple edges consecutively without stopping, see Chapter 2.6.3 (The Action Dependency Graph (ADG) framework). To model this behavior in simulation, clothoid curve traversal was implemented as follows: Whenever a shuttle i starts a new action a^i_j, it first checks if a^i_j represents a clothoid curve (Chapter 5.4 (Graph)). If, yes it recursively checks for the next action in its queue a^i_{j+1}. It continues until the first action does not represent a clothoid curve traversal anymore. Then all consecutive clothoid curve $c_j, ..., c_{j+n}$ for actions $a^i_j, ..., a^i_{j+n}$ are merged into one single combined clothoid curve C. Given C, TOPPRA is used to approximate the trajectory of the shuttle over C as described in Chapter 5.6.1 (Trajectory optimization).

Since C represents multiple edge traversals, notifying when the shuttle finished traveling over a single edge e_j, involves finding a sampled waypoint w_k in C that is closest to the target node of e_j. Querying the calculated trajectory $\mathrm{T}(w_k)$then provides the time t, when the traversal of e_j was finished.

6 Evaluation

This chapter provides a comprehensive evaluation of the proposed routing system. In Chapter 6.1 (Case Study: Lanemap - at Customer Warehouse), performance is evaluated in a real-world warehouse scenario. Chapter 6.2 (Comparison with Decentralized FCS Routing) offers a qualitative comparison to the decentralized approach currently used by FCS. The pros and cons of using a RHCRE-based routing system are further explored in Chapter 6.3 (Rolling Horizon Planning: Benefits and Challenges). Finally, the impact of the lanemap structure on routing success is discussed in Chapter 6.4 (Impact of Lanemap Structure on Routing Success).

6.1 Case Study: Lanemap - at Customer Warehouse

The basis for this case study poses a newly developed type of a fully autonomous warehouse content management and transport system that is solely based on AMRs. There shuttles drive over a dense road-network to transport goods between three main areas: These areas are:

1. **Delivery Stations**: Locations where shuttles deliver goods for human handling.
2. **Rack Stations**: Points where shuttles interact with racks to retrieve or store goods.
3. **Exchange Areas**: Zones used for transferring goods between different warehouse levels.

To maximize storage capacity racks are placed in a grid-like structure, as close together as possible. To ensure there is enough room for the navigation of shuttles, these racks are designed such that shuttles can move under them.

The dense physical environment of this Lanemap requires constant agent coordination. Additionally the grid-like environment offers many opportunities for agents to avoid each other by taking alternative routes. Traditional decentralized routing systems, have trouble with this kind of environment, as they are only coordinating agents reactively (if a problem occurs) (Chapter 3.4 (Drawbacks and Mitigation strategies)).

Specifically at the customer warehouse the main bottle neck was identified to be the area around the exchange stations. For this reasons this area was chosen to evaluate the centralized routing system proposed by this thesis (Chapter 4.2 (MAPD Routing System)). Figure 35 depicts this area. Highlighted in red are the delivery areas. Blue areas are rack stations. ***Note*** that these stations can only be reached from the same corridor as they are only connected via a small bidirectional lane. Therefore to move from one long corridor to another is only possible if no shuttle is occupying the overlapping station node associated with a rack. The exchange area is highlighted in green.

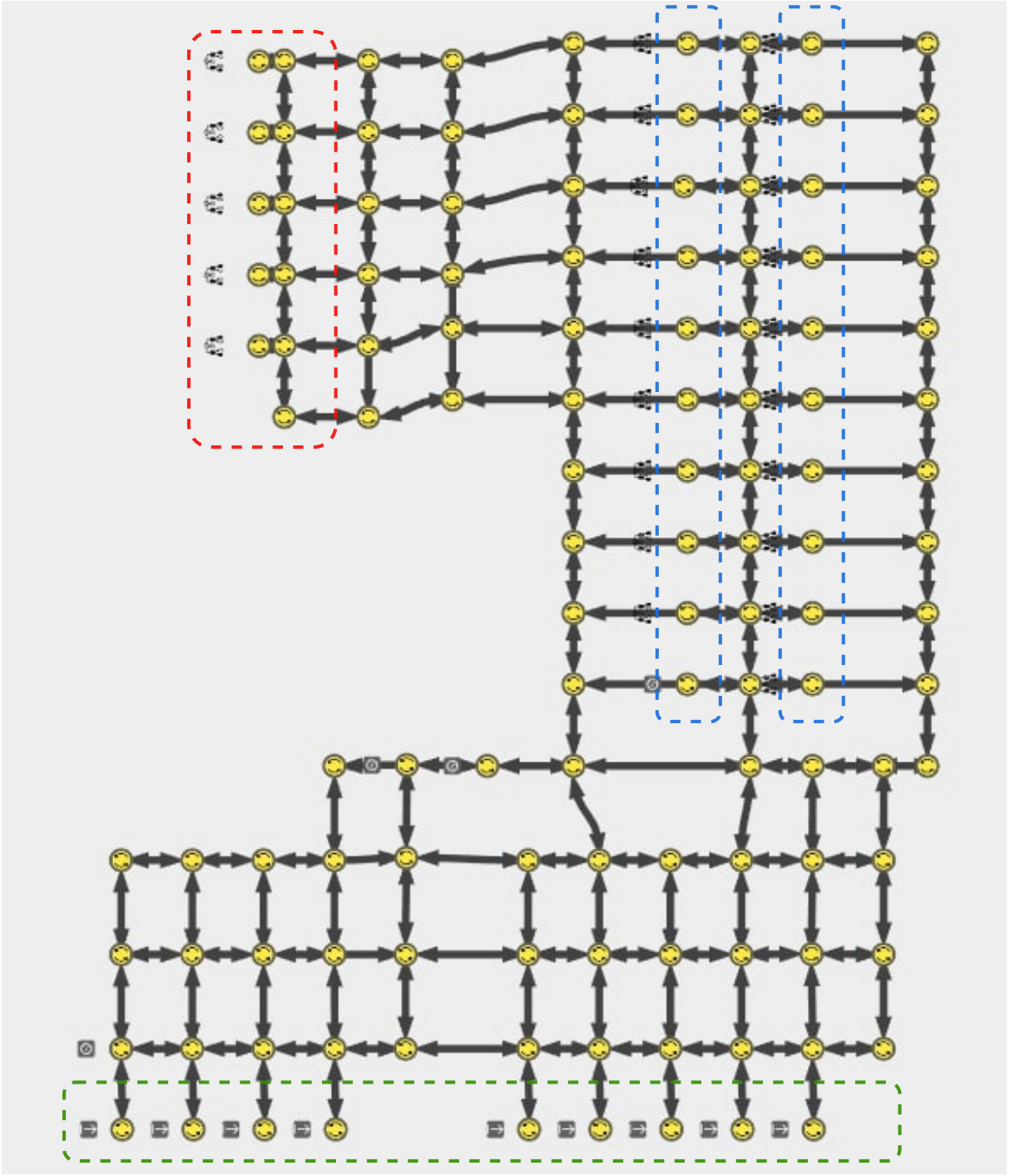


Figure 35: Lanemap of area around exchange stations at Customer Warehouse, used in the case study. The red area shows delivery stations, blue areas represent rack stations for box handling, and the green area represents exchange stations for inter-floor transfers.

6.1.1 Objective

In the chosen subset of the Lanemap, the aim was to have at least 10 agents continuously operating without interference.

6.1.2 Results

To evaluate the proposed system (Chapter 4.2 (MAPD Routing System)), a test setup with 40 orders was created. Each order contained 3 tasks with one of each station type (rack, delivery - and exchange station) assigned randomly. Overall, 120 tasks were performed by shuttles. Initially N shuttles were placed randomly on distinct Lanemap positions. Whenever a shuttle requested an order, a random one was assigned to it. The performance metrics used were **Makespan** (the overall runtime of the simulation) and **Flowtime** (the average time each shuttle took to complete a task), see Chapter 2.3.1.4 (MAPF - Objective Functions). 5 seconds was chosen as the planner timeout τ. For each N, the simulation was repeated 10 times.

Figure 36 illustrates these results.

In part **(a)**, the **Makespan** (in seconds) demonstrates a non-linear decreasing trend as the shuttle count **[N]** increases. Initially, the **Makespan** reduces as the shuttle count rises from 4 to 11, reflecting improved efficiency. However, this improvement slows, reaching a plateau at $N = 11$. Beyond 11, the **Makespan** increases again. This suggests that the overall system throughput is maximized at around 11 shuttles and no more shuttles should be added to this Lanemap.

In part **(b)**, the reason for this non-linear trend is clarified by the fact that the **Flowtime** increases with the number of shuttles.

Part **(c)** displays the simulation success rate over the number of shuttles. A single simulation is considered successful if all orders were completed by FCS Fake without a deadlock or consecutive planning failures. The success rate (%), therefore, indicates how many simulation runs were completed out of 10. First, the success rate remains stable up until $N = 12$, after which it sharply decreases. With 16 shuttles only 2 out of 10 simulations were successful. These failures primarily stem from two issues:

1. The PP-based planner timing out due to inherent limitations (see Chapter 2.4.1.1 (Optimality and Completeness)).
2. The planner finding a valid solution that leads to a deadlock in the next planning round, which will be further discussed in Chapter 6.3 (Rolling Horizon Planning: Benefits and Challenges)).

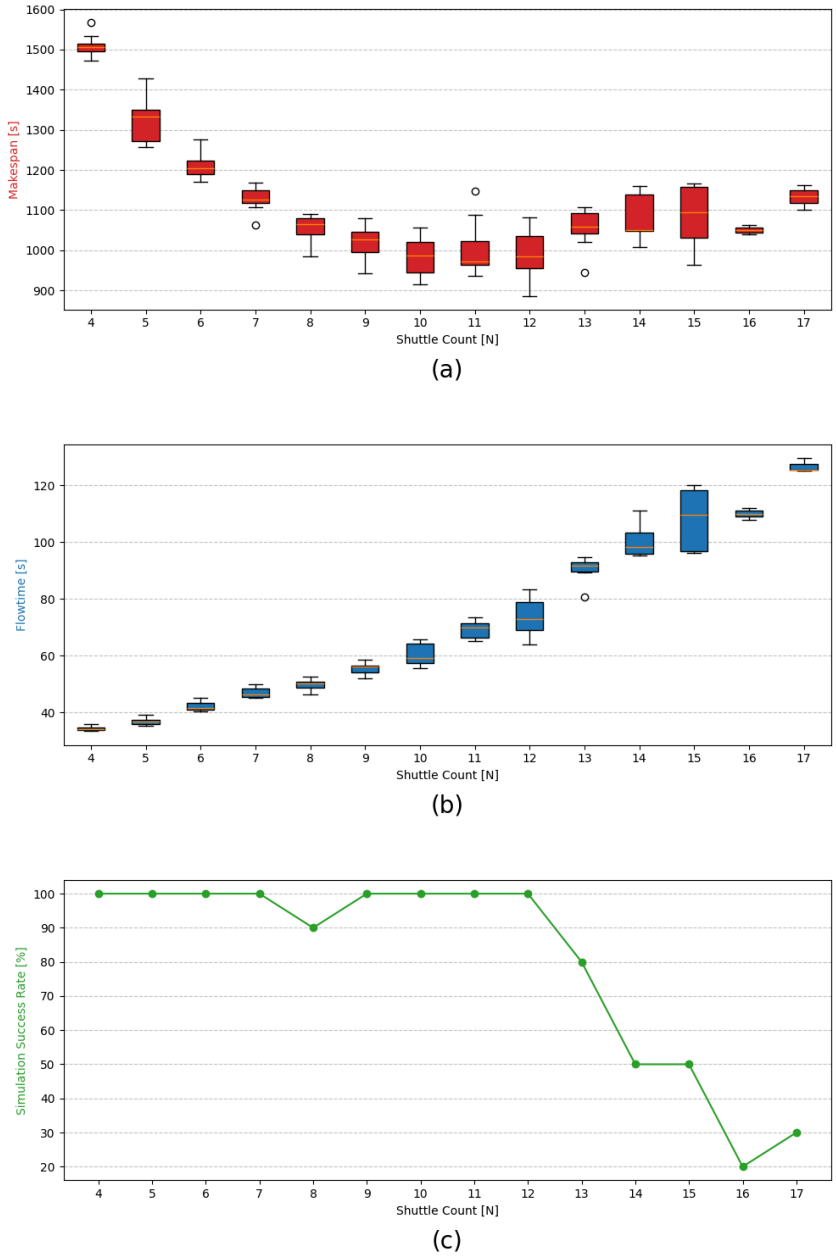


Figure 36: Customer Warehouse Case Study - Simulation results across different number of shuttles [N]: regarding (a) Makespan, (b) Flow Time and (c) Simulation Success Rate

Even in runs where a simulation failure occurred, the majority of planning rounds were usually successful. This suggests that only a few edge cases are responsible for the failures. Consider the simulation run depicted in Figure 37.

In this figure, the blue area represents the percentage of active shuttles over time. A shuttle is considered *active* when it is performing tasks or moving towards a task location. Conversely, a shuttle is considered *idle* when it is waiting for a task assignment or a valid plan to be computed. Initially - after the first planning run - all shuttles are active. After that shuttles periodically switch between active and idle. As visible by the red dotted line, overall ~60% shuttles are active at any given time. ***Note*** that at the end the shuttle activity reaches 100%,

as shuttles without active orders are sent to random locations to not block goals of other still busy shuttles.

Green and red vertical intervals correspond to successful and failed replanning attempts, respectively. Although most planning attempts were successful, there is a notable concentration of failed planning rounds during a specific period between 400 and 600 seconds. These failures coincide with a time of high congestion in the simulation. When congestion is high, multiple shuttles may attempt to occupy the same routes or regions of the environment, which increases the likelihood of deadlocks or planning conflicts. In this case, even though the many planning failures occurred, subsequent planning rounds remained solvable, and the system was able to continue. However, if by chance an unsolvable situation occurs that is not resolved by subsequent plans, after the shuttles continue on their remaining path, a simulation failure occurs.

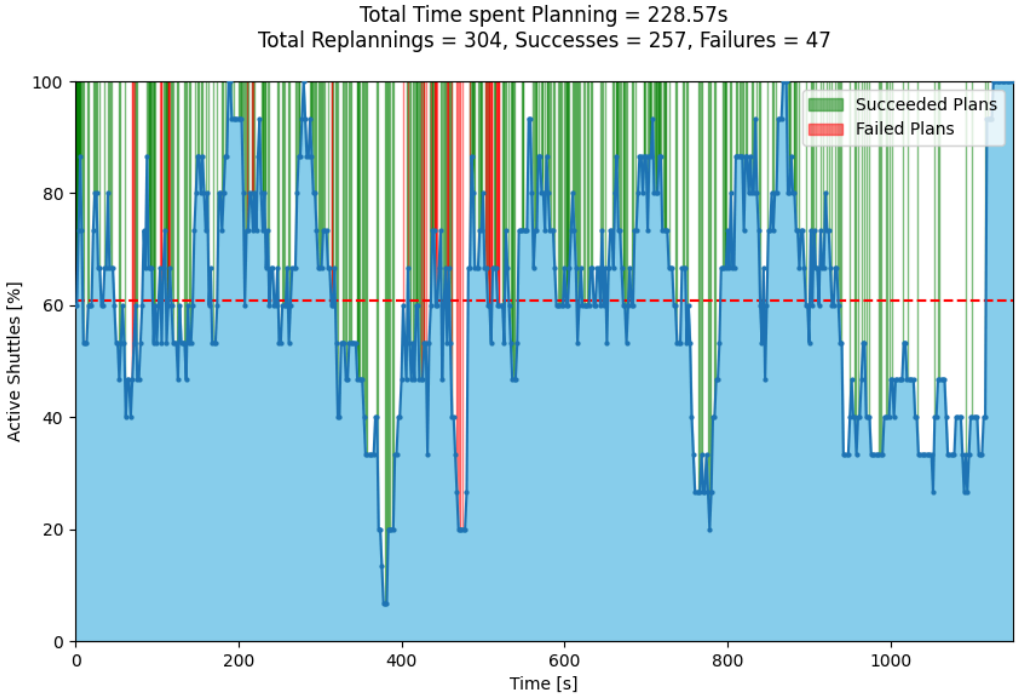


Figure 37: Shuttle Activity and Planning summary across simulation run for 15 agents. The blue area shows active shuttle percentages, while green and red intervals show successful and failed replanning attempts. The red dashed line represents the overall average active shuttle percentage

The simulation's ability to recover from these situations, or lack thereof, significantly affects the overall success rate. The observed failed attempts are thus a result of localized issues rather than a system-wide failure. These critical points of failure highlight potential areas for improvement, which will be further discussed in subsequent chapters, such as Chapter 6.3 (Rolling Horizon Planning: Benefits and Challenges) and Chapter 6.4 (Impact of Lanemap Structure on Routing Success).

6.1.3 Impact of Agents on Planner Performance

This section analyzes how the number of agents (shuttles) impacts the performance of the planner. The goal is to evaluate the scalability of the system as the number of shuttles increases and how different components of the planning process (e.g., single priority rounds, SIPP planning) contribute to the overall runtime. Figure 38 illustrates the runtime of the planner as the number of shuttles increases from 4 to 17. As the number of agents increases each component of the planning process scales differently. Starting from the lowest level (**blue**), the average duration of a single SIPP planning round remains relatively constant, with a slight increase as the number of agents rises. This is expected, as the SIPP planner is only responsible for planning the path of a single agent and is only affected by the number of agents insofar as more congestion leads to longer paths for every individual shuttle.

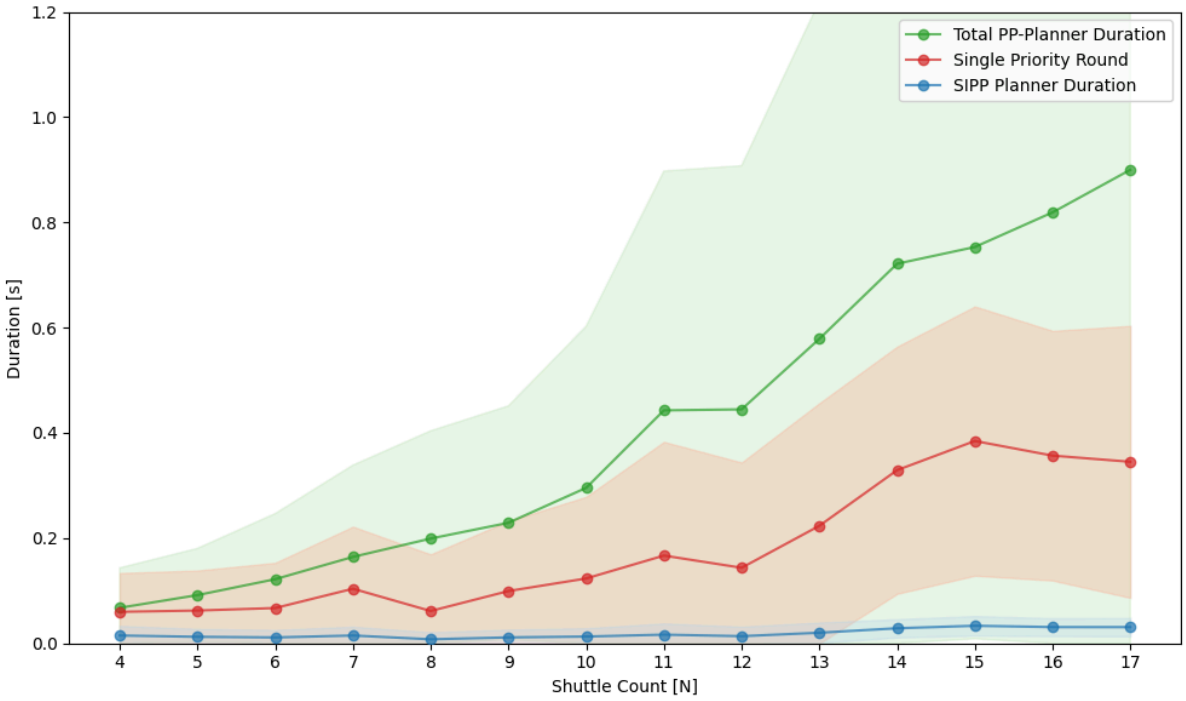


Figure 38: Planner Scalability [s] per number of shuttles [N]. (green): The average single-shot MAPF planner duration, consisting of $1 - k$ priority planning rounds. (red): The average priority planning round duration. (blue): The average SIPP single-agent planner duration.

Given a priority ordering P, a single priority round (**red**) executes the underlying single agent planner for each agent in P once, which (as expected) shows a linear trend with the number of agents.

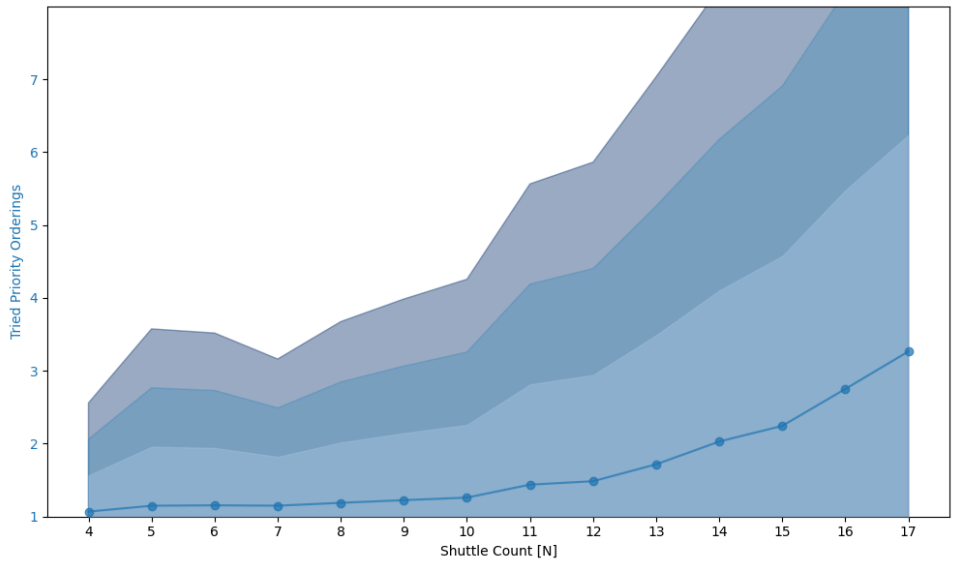


Figure 39: The average number of priority ordering retries per shuttle count [N]. The blue line represents the average retries and the shaded regions show 1, 2 and 3 - sigma std. deviations.

The total duration of the single-shot MAPF planner (**green**) is the sum of all priority rounds. As the number of agents increases, the total duration of the planner rises stronger than that of a single priority round. As depicted in Figure 39, this is due to the fact that the planner on average has to try more priority orderings to find a valid solution.

6.1.4 Discussion of Results

These results effectively show that the objective of 10 agents was surpassed, as the proposed system supports robust operation with up to 12 shuttles. However, its limitations become evident with larger numbers, as discussed earlier. Subsequent chapters will explore specific scenarios and edge cases that challenge the system's effectiveness.

6.2 Comparison with Decentralized FCS Routing

It is important to note that while a comparison between the proposed MAPD routing (Chapter 4.2 (MAPD Routing System)) system and the current decentralized FCS routing (Chapter 3.3.3 (Employed Decentralized Fleet coordination)) is provided, these systems are not directly comparable. The main reason for this lies in the fundamental differences in the simulation environments and the complexity of the real-world operations handled by FCS (see Chapter 4.1 (Simulation (FCS Fake))). For instance, in the real FCS, whenever a shuttle is instructed to move from one node to another, its response is governed by an internal behavior tree and numerous additional complexities that are not modeled in FCS Fake. These complexities preclude a direct quantitative comparison, particularly in terms of execution time. In fact, the inherent complexity of the FCS system necessitated the development of a bare-boned test environment in the first place.

Given these limitations, this thesis focuses on providing a qualitative comparison between the two systems.

6.2.1 Lanemap Structure

In the context of the decentralized FCS routing system, the design of the Lanemap is critical for effective operation. (Chapter 3.4 (Drawbacks and Mitigation strategies)).

Traditionally, the Lanemap for FCS has often been designed to resemble a round course with a single direction of travel. This approach naturally reduces conflicts in traffic flow but has several disadvantages:

1. **Inefficient Use of Space**: Physical space is not utilized efficiently, leading to wasted potential storage and operational areas.
2. **Longer Paths**: Shuttles are required to follow longer paths than necessary, which reduces overall efficiency.

Switching to a bidirectional Lanemap improves throughput by allowing shuttles to avoid each other with much less overhead.

In contrast to the bidirectional setup depicted in Figure 36, the Lanemap used by FCS is depicted in Figure 40. When comparing these two Lanemaps, one may notice that the areas in front of the exchange (green) and delivery stations (red) contain a round course like structure where shuttles can only move in and out in one direction. Additionally most lanes between the racks have been converted to unidirectional "high-ways". Areas where bidirectional lanes are necessary, or where excessive congestion could lead to system failure, are annotated with N-robot areas (Chapter 3.4.2 (N-Robot area)). While this design is functional, it requires significant manual effort to set up and maintain. In contrast, the proposed routing system (Chapter 4.2 (MAPD Routing System)) naturally accommodates bidirectional lanes, allowing for more efficient use of physical space and shorter paths for shuttles during operation.

Collaboration is inherent to this design, which is expected to not only reduce manual setup and maintenance effort but also improve overall system throughput.

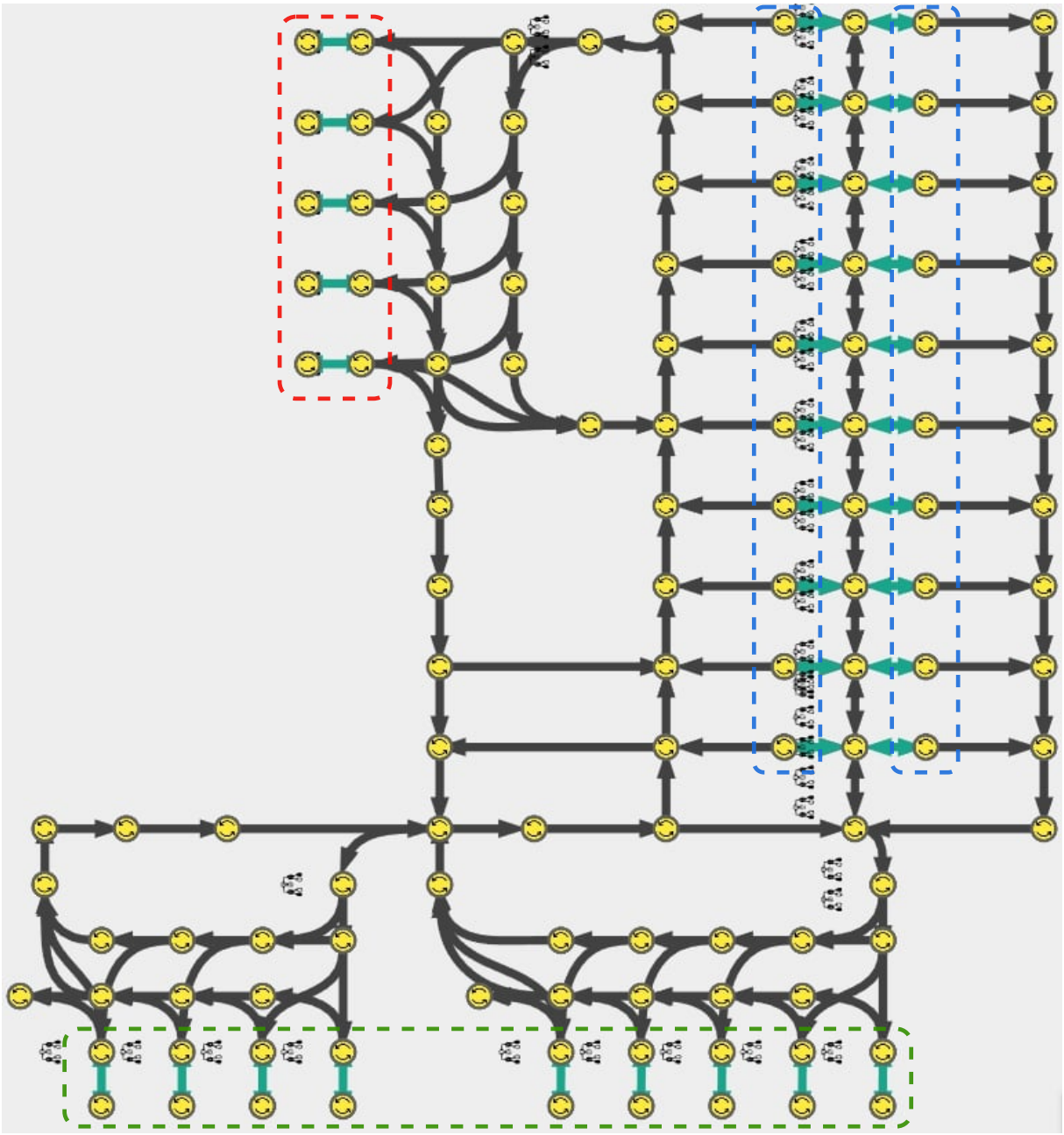


Figure 40: Lanemap of area around exchange stations at Customer Warehouse for the decentralized routing system.

6.2.2 Utilizing corridors

To further understand how the proposed system allows for a more efficient use of space, consider the scenario depicted in Figure 25: As mentioned in Chapter 3.4.3 (Deadlock resolution), the decentralized FCS routing system has trouble with these scenarios and can run into a "life-lock". The proposed collaborative routing system finds plans considering paths for both agents. Thus, if shuttles face each other head on, it will always naturally send one of them into the corridor and thus preemptively avoid any deadlock situations.

6.2.3 Random Task Assignment Deadlock

While most deadlock situations the FCS faces can be avoided by its existing mitigation strategies (Chapter 3.4 (Drawbacks and Mitigation strategies)), there is a class of deadlocks that are inherently hard to avoid with a decentralized system.

As outlined in Chapter 3.2.2 (Order and Task Allocation), the FCS system assigns tasks independently of routing. Without a collaborative routing strategy, this can lead to deadlock situations where tasks are assigned to shuttles based on proximity alone, resulting in infeasible routing. An example of this is shown in Figure 41.

In the depicted scenario, two shuttles navigate a round-course-like Lanemap with strategically placed crossover lanes to facilitate efficient traffic flow. The teal shuttle S_t receives a task to pick up a box from the station above just as the violet shuttle S_v is waiting for S_t to clear the crossover lane it needs. However, S_t cannot move to its task because it is blocked by S_v, creating a deadlock where both shuttles are stuck, unable to proceed.

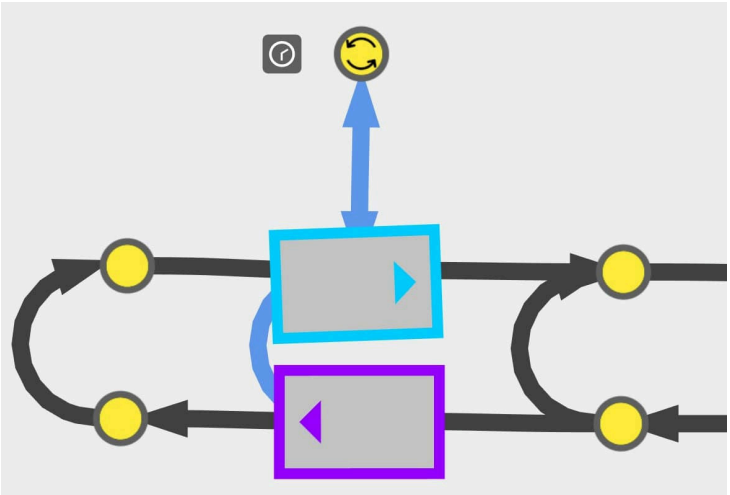


Figure 41: FCS - Deadlock scenario caused by random task assignment. Shuttle S_v (violet) is waiting for shuttle S_t , while S_t (teal) can not rotate on the spot because of S_v.

Note that Figure 23 (1) also falls into this class of deadlocks.

A collaborative routing system, such as the one proposed in this thesis, would prevent this deadlock by moving S_t out of the way preemptively.

6.2.4 Limitations of Prioritized Planning

As described in Chapter 4.2.2 (MAPF Planner), our routing system uses a prioritized planning approach internally. While this works well in a lot of cases, there are scenarios where it fails (see Chapter 2.4.1.1 (Optimality and Completeness)). In general prioritized planning does not work if it requires a higher prioritized agent to move out of the way.

Figure 42 illustrates a scenario where two shuttles operate within a shared environment featuring one fast lane and two slower lanes on either side. The fast lane in the middle allows for

quick movement when unoccupied, while the side lanes serve as avoidance routes when two shuttles approach each other.

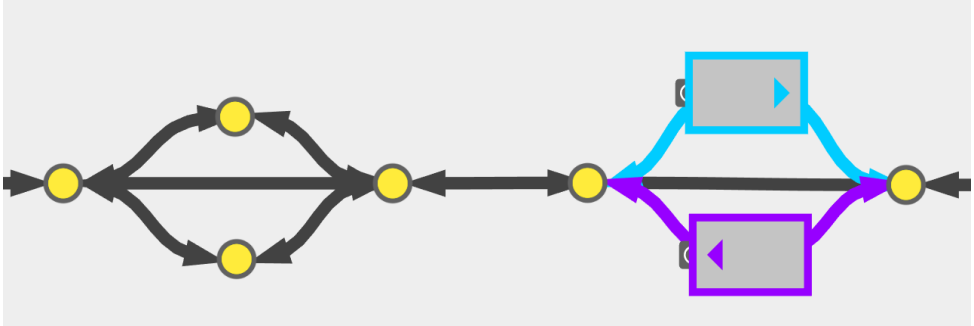


Figure 42: Collaborative avoidance areas at customer warehouse. Manually defined routines allow complex avoidance maneuvers.

In this setup, if two shuttles face each other head-on (with one on each side of the avoidance area), a manual navigation routine is triggered. One shuttle is directed to the upper node, and the other to the lower, allowing both to bypass each other safely and for traffic to resume normally. However, there is not enough space to allow one shuttle to use the the fast lane and another one of the adjacent lanes at the same time.

This configuration works efficiently under manual control but highlights a significant limitation of a Prioritized Planning (PP) system. There, a higher prioritized shuttle would always take the fast lane in the middle and block the whole avoidance area. This would prevent shuttles from using the avoidance area and lead to a deadlock.

One way to solve this issue is to use a Conflict Based Search (CBS) -based planner instead (Chapter 2.4.5 (Continuous-time Conflict-based Search (CCBS))). Specifically, Continuous-Time Conflict-Based Search (CCBS) can serve as a direct replacement for our existing PP based planner The continuous nature of CCBS allows for the integration of the same enhancements that were applied to PP, including precomputed overlap set collision lookups and seamless integration into the RHCRE framework (Algorithm 6).

Although CBS-based methods are complete and optimal, they tend to scale exponentially with the number of conflicts. While standard CBS performs efficiently with up to several hundred agents (Li, Ruml and Koenig, 2021), the FCS operates on a graph where orientation is explicitly modeled (Chapter 3.3 (Path Planning and Fleet coordination)). This setting can result in a much higher number of conflicts, raising concerns about how well CCBS will perform in this specific context.

6.3 Rolling Horizon Planning: Benefits and Challenges

As already described in Chapter 2.6.6 (Planning for a windowed time horizon (RHCR)), using a rolling horizon approach makes lifelong MAPF planning well defined and more efficient. This approach offers unique benefits and challenges that are worth discussing in more detail.

1. **Efficiency - Optimality Trade-off**

When a rolling horizon w is used, the MAPF planner only needs to avoid collisions within $[0, w]$. This allows individual planning rounds to finish much faster, significantly increasing overall efficiency (Li *et al.*, 2021). The inherent downside of this approach is that it does not account for collisions that might occur after the planning horizon, which can lead to situations where no progress is made (life-lock).

Consider the example depicted in Figure 25. Suppose a new planning round starts with the current shuttle positions as the starting points and the leftmost and rightmost nodes as the goals for the teal and violet shuttles, S_t and S_v. If the rolling horizon w is shorter than the time it takes for S_t to reach the avoidance corner, the planner may fail to send S_t to the avoidance corner in time. Depending on the length of the corridor, this can result in shuttles repeatedly moving back and forth without making any progress. (Li *et al.*, 2021) discusses this issue and suggests dynamically increasing the planning horizon until shuttles begin making progress again.

2. **Robustness to Congestion - Deadlock between Planning Rounds**

As already discussed in Chapter 2.6.6 (Planning for a windowed time horizon (RHCR)), a significant advantage of using a rolling horizon is that planning does not fail outright, even if no valid MAPF solution exists (e.g., when two shuttles have the same goal) or if an area is too congested for the shuttles involved to make progress. In a lifelong context, such situations typically resolve when a blocking shuttle moves elsewhere. Using RHCRE allows the planner to find plans even if some shuttles are unable to progress toward their goals.

The downside is that shuttles might be sent into positions that will lead to a deadlock in the next planning round. Consider the example depicted in Figure 43. (1): At the beginning of the planning round both shuttles S_g and S_b have the same target. Without RHCR, this situation would lead to a planning failure. (2): With RHCR the planner will find solutions that allow S_g to progress as far as possible until the end of the planning horizon. After h collisions are not considered anymore. From the MAPF planner perspective both shuttles can finish at their goal position. In the context of RHCRE, the last action that gets sent to S_g is to wait at the nearest possible node from its goal until the end of the planning horizon. In a lifelong situation, this behavior is desired. In the context of this thesis however, (planning on a orientational routing graph with large agents) this can lead to the above mentioned edge case. On the routing graph

level, S_g effectively moved into a dead end, because rotation on the spot is not possible from there. This class of deadlocks is denoted from here on as **orientation dependent deadlock**.

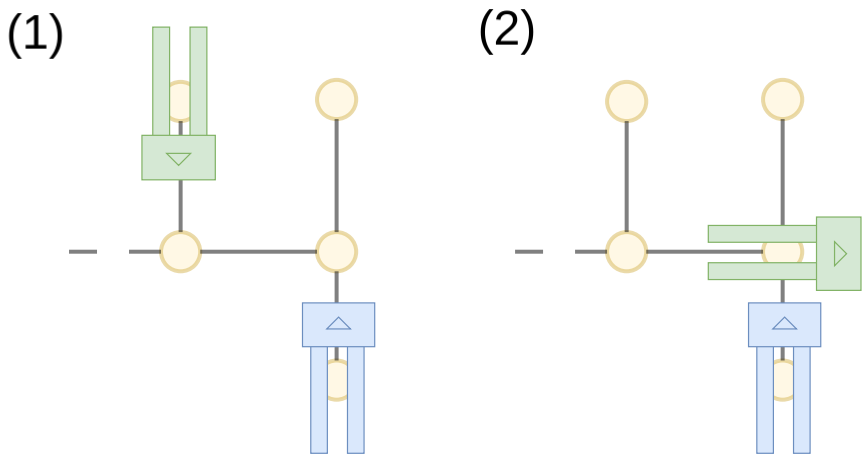


Figure 43: Example scenario of a deadlock between planning rounds, due to Rolling Horizon Planning (RHCR), (1): At the beginning of the planning round, both the green shuttle S_g and the blue shuttle S_b have the same target. (2): Due to RHCR, S_g finishes its plan at a position where a deadlock occurs at the start of the next planning round.

6.4 Impact of Lanemap Structure on Routing Success

The potential for orientation-dependent deadlocks (as depicted in Figure 43) is directly influenced by the structure of the Lanemap. In classical MAPF, an agent at node A can always move to any of its neighbors A' if A' is free. However, with large agents this is not always the case,especially not in the context of FCS where the Lanemap directly represents the motion trajectories that large agents should follow (Chapter 3.1.2 (Lanemap)). ***Note:*** graph nodes, where the above statement holds will further be referred to as ***maneuverable***.

For **arbitrary** clothoid curve roadmaps, deadlock corner cases cannot generally be avoided. Thus, it can be concluded that the robustness of any routing system heavily depends on the underlying graph structure.

This is exemplified in the scenario depicted in Figure 44. There, complex shuttle maneuvers are directly modeled within the lanemap. To navigate to the pickup station below (highlighted in green), a shuttle can either come from the blue node or the violet one. Exiting this area is only possible via the blue node. An N(1)-robot area was placed over all (red) lanes. Without it, deadlock free navigation could not be ensured.

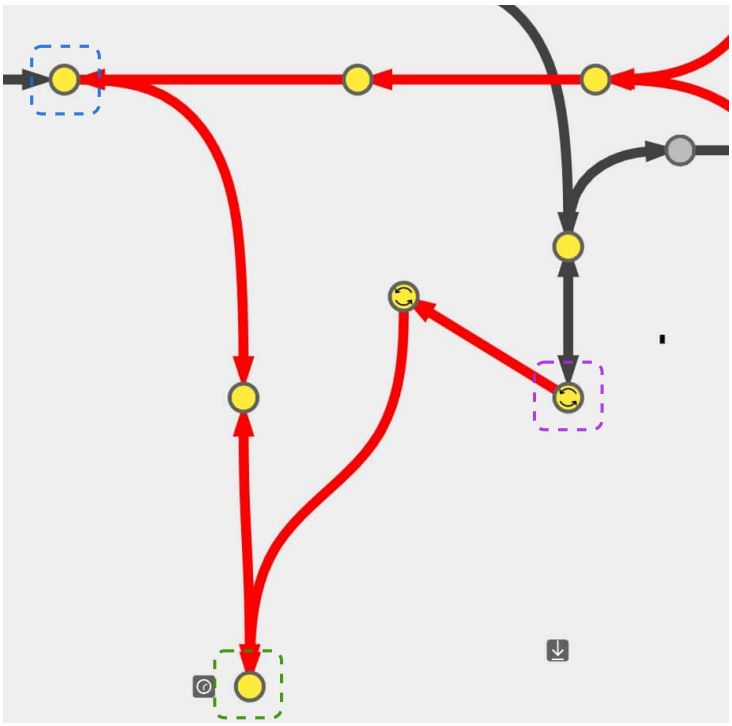


Figure 44: Lanemap modeling complex shuttle trajectories. Shuttles navigate to the green pickup station from either the blue or violet nodes. Exit is only possible via the blue node. Red lanes form an N(1)-robot area, preventing deadlocks.

7 Conclusion & Future Work

7.1 Conclusion

This thesis presents the development of a novel centralized Multi-Agent Pickup and Delivery (MAPD) routing system designed for large agents navigating clothoid curve roadmaps. The core objective was to replace the decentralized coordination strategy currently used in the Fleet Control System (FCS) by addressing inherent limitations related to scalability, resilience to delays, and operational continuity within a dynamic warehouse environment.

The proposed system effectively integrates recent advances in MAPF and its extension MAPD, specifically on euclidean graphs. By combining the idea of pre-computing Continuous Time Conflicts (CTCs) with the Rolling Horizon Collision-Resolution and Execution (RHCRE) framework, in an orientation graph context where in-place rotations are modeled explicitly, the system demonstrates the effectiveness of a centralized approach for managing the coordination of large agents in continuous space.

Despite these successes, the thesis acknowledges the limitations of the proposed approach, particularly concerning the incompleteness of a Prioritized Planning (PP)-based scheme. While using an orientational graph representation simplifies routing for large agents, it also introduces more complex deadlock scenarios that may not be solvable without considering the physical structure of the routing graph. Mitigation strategies for these challenges have been outlined, including (i) combining PP with CCBS, (ii) leveraging topological graph annotations, and (iii) generalizing the concept of N-robot areas into a hierarchical routing graph structure. Additionally, future research avenues for centralized planning systems have been proposed, such as the development of dynamic traffic avoidance strategies—an opportunity traditionally unavailable to systems that lack a centralized controller with access to full system information.

In conclusion, the work presented in this thesis demonstrates that centralized schemes not only offer theoretical advantages over decentralized ones (like the FCS) but, with recent advances in MAPF and MAPD, are now ready for production use. This marks a significant shift, indicating that centralized paradigms are poised to replace decentralized strategies in the near future.

7.2 Future Work

This chapter discusses potential directions for future research based on the findings of this thesis. Chapter 7.2.1 (Combining PP and CBS) explores the combination of PP and CBS into a scalable solver that overcomes limitations faced by PP-based techniques. Chapter 7.2.2 (Topological Graph Annotation) reviews recent progress in including topology information to guide the CBS search, further reducing the planning effort. Chapter 7.2.3 (Generalizing N-robot Areas) presents our idea of generalizing the concept of N-robot areas into a hierarchical routing

graph structure. Finally, Chapter 7.2.4 (Dynamic Traffic Avoidance) examines how a centralized planning scheme can be extended to include dynamic traffic awareness.

7.2.1 Combining PP and CBS

As already stated in Chapter 6.2.4 (Limitations of Prioritized Planning)

1. prioritized planning has inherent limitations that can be addressed by using a CBS-based planner.
2. CCBS can be directly integrated into our proposed routing system (Chapter 4.2.2 (MAPF Planner)).

Still it remains an open question how well CCBS scales within our MAPD routing system. The authors of CCBS (Andreychuk *et al.*, 2021) mention that much of the algorithms runtime is due to complex geometric collision checking. The proposed approach leverages precomputed overlap sets (Chapter 4.2.2.2 (MAPF Planner - Algorithm)) for collision checking and therefore does not face this issue. On the other hand routing is performed on a graph structure where orientational transitions are modeled explicitly. Since CBS based methods scale exponentially with the number of conflicts this could mitigate the advantage of fast collision checking.

If CCBS proves to scale reasonably within an orientation graph setting, one further direction that could be taken is to combine PP and CCBS into a hybrid planner. The authors of (Park *et al.*, 2023) propose Conflict-Based Search with Partitioned Groups of Agents (CCBS-PGA). There, agents are dynamically partitioned into groups. Within groups, agents are planned with CCBS, while between groups PP is used. They claim that this approach overcomes some of the limitations inherent to PP while still remaining scalable for real-world operation.

7.2.2 Topological Graph Annotation

Another potential direction to improve the scalability of CCBS is to include topological information in the routing graph. The authors of (Song, Na and Yu, 2023) propose annotating specific parts of the routing graph with topological information, which can significantly enhance performance in scenarios where CBS-based methods typically struggle. Their approach groups nodes that are part of a corridor into a topological area. When two agents attempt to enter the same area from opposite directions, the planner considers a corridor-conflict rather than a series of individual node-conflicts. This approach greatly reduces the number of conflicts, as in native CBS, a node-conflict would be considered for each node within the corridor. In a sense, using topological annotation generalizes the idea illustrated in Figure 42 , where the optimal routing behavior for specific regions is manually defined in advance.

7.2.3 Generalizing N-robot Areas

As discussed in Chapter 6.4 (Impact of Lanemap Structure on Routing Success), continuous routing graphs cannot generally be guaranteed to be deadlock-free, and one way to mitigate this is by using something like N-robot areas (Chapter 3.4.2 (N-Robot area)).

Depending on the complexity and density of the Lanemap structure, it may not always be practical to consider certain entities (nodes or edges) as distinct when solving a MAPF problem. For instance, Figure 44 illustrates an example where multiple nodes and edges are grouped into a single N-robot area to ensure deadlock-free navigation. From a routing perspective, it would be simpler and more efficient to treat the entire N-robot area as a single node or unit within the routing graph.

This abstraction would naturally lead to the the concept of a hierarchical planning graph. A MAPF solver would then operate on the "area" level, while a lower-level path planner could find SAPF paths within that area. Areas could either be single *maneuverable* nodes or big N-robot areas. This way, Lanemaps could still be used for complex motion planning, while routing would be simplified and more efficient.

7.2.4 Dynamic Traffic Avoidance

Beyond exploring ways to mitigate the inherent limitations of our MAPD routing system, it is crucial to recognize the broader opportunities that a centralized planner provides. With all agent information managed by a single controller, a centralized approach offers numerous avenues for extension that are not available to decentralized systems. Centralized access to comprehensive data enables the implementation of advanced schemes and strategies, such as dynamic traffic avoidance. The authors of (Chen *et al.*, 2024) propose a lifelong MAPF planning framework where edge costs are dynamically adjusted based on the amount of traffic occurring in that area. They show that this effectively distributes traffic across the entire network, significantly reducing congestion.

Bibliography

Ali, Z. A. and Yakovlev, K. (2023) *Safe Interval Path Planning With Kinodynamic Constraints*. arXiv. Available at: https://doi.org/10.48550/arXiv.2302.00776

Andreychuk, A. and Yakovlev, K. (2018) "Two Techniques That Enhance the Performance of Multi-robot Prioritized Path Planning"

Andreychuk, A. *et al.* (2021) *Improving Continuous-time Conflict Based Search*. arXiv. Available at: https://doi.org/10.48550/arXiv.2101.09723

Andreychuk, A. *et al.* (2022) "Multi-agent pathfinding with continuous time," *Artificial Intelligence*, 305, p. 103662–103663. Available at: https://doi.org/10.1016/j.artint.2022.103662

Atzmon, D. *et al.* (2018) "Robust Multi-Agent Path Finding," *Proceedings of the International Symposium on Combinatorial Search*, 9(1), pp. 2–9. Available at: https://doi.org/10.1609/socs.v9i1.18445

Chen, Z. *et al.* (2024) "Traffic Flow Optimisation for Lifelong Multi-Agent Path Finding," *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(18), pp. 20674–20682. Available at: https://doi.org/10.1609/aaai.v38i18.30054

Dechter, R. and Pearl, J. (1985) "Generalized best-first search strategies and the optimality of A*," *Journal of the ACM*, 32(3), pp. 505–536. Available at: https://doi.org/10.1145/3828.3830

Diestel, R. (2017) "The Basics," *Graph Theory*. Berlin, Heidelberg: Springer. Available at: https://doi.org/10.1007/978-3-662-53622-3_1

Erdmann, M. and Lozano-Pérez, T. (1987) "On multiple moving objects," *Algorithmica*, 2(1), pp. 477–521. Available at: https://doi.org/10.1007/BF01840371

Gregoire, J., Čáp, M. and Frazzoli, E. (2017) "Locally-optimal multi-robot navigation under delaying disturbances using homotopy constraints," *Springer US* [Preprint]. Available at: https://dspace.mit.edu/handle/1721.1/116601 (Accessed: July 23, 2024)

Hart, P. E., Nilsson, N. J. and Raphael, B. (1968) "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, 4(2), pp. 100–107. Available at: https://doi.org/10.1109/TSSC.1968.300136

Hoenig, W. *et al.* (2019) "Persistent and Robust Execution of MAPF Schedules in Warehouses," *IEEE Robotics and Automation Letters*, 4(2), pp. 1125–1131. Available at: https://doi.org/10.1109/LRA.2019.2894217

Hoenig, W. *et al.* (2016) "Multi-Agent Path Finding with Kinematic Constraints," *Proceedings of the International Conference on Automated Planning and Scheduling*, 26, pp. 477–485. Available at: https://doi.org/10.1609/icaps.v26i1.13796

Hoenig, W. *et al.* (2018) "Trajectory Planning for Quadrotor Swarms," *IEEE Transactions on Robotics*, 34(4), pp. 856–869. Available at: https://doi.org/10.1109/TRO.2018.2853613

Kasaura, K., Nishimura, M. and Yonetani, R. (2022) "Prioritized Safe Interval Path Planning for Multi-Agent Pathfinding With Continuous Time on 2D Roadmaps," *IEEE Robotics and Automation Letters*, 7(4), pp. 10494–10501. Available at: https://doi.org/10.1109/LRA.2022.3187265

Kohout, P. (2024) *AUTHOR WAS UNABLE TO PROVIDE ME A TITLE IN TIME*

LaValle, S. M. (2006) *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press

Li, J. *et al.* (2020) "New Techniques for Pairwise Symmetry Breaking in Multi-Agent Path Finding," *Proceedings of the International Conference on Automated Planning and Scheduling*, 30, pp. 193–201. Available at: https://doi.org/10.1609/icaps.v30i1.6661

Li, J., Ruml, W. and Koenig, S. (2021) *EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding*. arXiv. Available at: https://doi.org/10.48550/arXiv.2010.01367

Li, J. *et al.* (2021) *Lifelong Multi-Agent Path Finding in Large-Scale Warehouses*. arXiv. Available at: http://arxiv.org/abs/2005.07371 (Accessed: July 1, 2024)

Ma, H. *et al.* (2019) "Searching with Consistent Prioritization for Multi-Agent Path Finding," *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(1), pp. 7643–7650. Available at: https://doi.org/10.1609/aaai.v33i01.33017643

Ma, H. *et al.* (2017) *Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks*. arXiv. Available at: http://arxiv.org/abs/1705.10868 (Accessed: February 14, 2024)

Moravec, H. and Elfes, A. (1985) "High resolution maps from wide angle sonar," in *1985 IEEE International Conference on Robotics and Automation Proceedings. 1985 IEEE International Conference on Robotics and Automation Proceedings*, pp. 116–121. Available at: https://doi.org/10.1109/ROBOT.1985.1087316

Park, C. *et al.* (2023) "Conflict-Based Search with Partitioned Groups of Agents for Real-World Scenarios," in *2023 20th International Conference on Ubiquitous Robots (UR). 2023 20th International Conference on Ubiquitous Robots (UR)*, pp. 986–992. Available at: https://doi.org/10.1109/UR57808.2023.10202470

Pham, Q.-C. (2014) "A General, Fast, and Robust Implementation of the Time-Optimal Path Parameterization Algorithm," *IEEE Transactions on Robotics*, 30(6), pp. 1533–1540. Available at: https://doi.org/10.1109/TRO.2014.2351113

Phillips, M. and Likhachev, M. (2011) "SIPP: Safe interval path planning for dynamic environments," in *2011 IEEE International Conference on Robotics and Automation. 2011 IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China: IEEE, pp. 5628–5635. Available at: https://doi.org/10.1109/ICRA.2011.5980306

Pinedo, M. (2008) *Scheduling: Theory, Algorithms, and Systems*. 4th ed. New York: Springer, p. 698–699. Available at: https://doi.org/https://doi.org/10.1007/978-3-031-05921-6

Pygame-Community (2024) *Pygame: Python Game Development*. Available at: https://pygame.readthedocs.io/en/latest/1_intro/intro.html

Russell, S. J., Norvig, P. and Davis, E. (2010) *Artificial intelligence: a modern approach*. 3rd ed. Upper Saddle River: Prentice Hall (Prentice Hall series in artificial intelligence)

Scheuer, A. and Fraichard, T. (1997) "Continuous-curvature path planning for car-like vehicles," in *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS '97. Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS '97*, pp. 997–1003. Available at: https://doi.org/10.1109/IROS.1997.655130

Sharon, G. *et al.* (2012) "Conflict-Based Search For Optimal Multi-Agent Path Finding," *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1), pp. 563–569. Available at: https://doi.org/10.1609/aaai.v26i1.8140

Silver, D. (2005) "Cooperative Pathfinding," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 1(1), pp. 117–122. Available at: https://doi.org/10.1609/aiide.v1i1.18726

Song, S., Na, K.-I. and Yu, W. (2023) "Anytime Lifelong Multi-Agent Pathfinding in Topological Maps," *IEEE Access*, 11, pp. 20365–20380. Available at: https://doi.org/10.1109/ACCESS.2023.3249471

Stern, R. *et al.* (2019) *Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks*. arXiv. Available at: http://arxiv.org/abs/1906.08291 (Accessed: October 23, 2023)

Symons, J. (2017) *CBS Conflict Based Search Algorithm*. Available at: https://www.youtube.com/watch?v=FnrZyL6965o

V.P. Kostov, E. D.-K. (1992) "Institut national de recherche en informatique et en automatique," *Bulletin of Sociological Methodology/Bulletin de Méthodologie Sociologique*, 37(1), pp. 55–57. Available at: https://doi.org/10.1177/075910639203700105

Varambally, S., Li, J. and Koenig, S. (2022) "Which MAPF Model Works Best for Automated Warehousing?," *Proceedings of the International Symposium on Combinatorial Search*, 15(1), pp. 190–198. Available at: https://doi.org/10.1609/socs.v15i1.21767

Yu, J. and LaValle, S. (2013) "Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs," *Proceedings of the AAAI Conference on Artificial Intelligence*, 27(1), pp. 1443–1449. Available at: https://doi.org/10.1609/aaai.v27i1.8541

Zhang, K. *et al.* (2024) "A priority-based hierarchical framework for \textit{k} -robust multi-agent path finding," *IEEE Transactions on Intelligent Vehicles*, pp. 1–11. Available at: https://doi.org/10.1109/TIV.2024.3355423

Zinoviev, D. (2024) *Discrete Event Simulation: It's Easy with SimPy!*. arXiv. Available at: https://doi.org/10.48550/arXiv.2405.01562

Acronyms

***ADG* – Action Dependency Graph**: A robust MAPF plan execution framework that models time as implicit in the action precedence order, thus allowing for uncertainty and fault-robust plan execution in real-world scenarios. viii, ix, 31, 32, 33, 34, 35, 36, 37, 39, 52, 54, 56, 64, 65, 66, 67, 68, 71

***AMR* – Autonomous Mobile Robot**. 1, 2, 15, 17, 32, 36, 40, 75

***CBS* – Conflict Based Search**: A complete and optimal but exponential two-level search algorithm for the MAPF problem. viii, 24, 25, 27, 29, 32, 34, 85, 89, 90

***CCBS* – Continuous-Time Conflict-Based Search**: An extension of CBS that works with continuous time. viii, 27, 28, 29, 32, 85, 89, 90

***CCBS-PGA* – Conflict-Based Search with Partitioned Groups of Agents**: A CBS - PP hybrid that partitions agents into groups. Within groups agents are planned with CBS, while between groups PP is used. 90

CLion: A cross-platform (C++) IDE by © JetBrains. 69

C++: A general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language. 69, 72

***CT* – Constraint Tree**: A binary tree that represents the constraints between agents in CBS. 24, 25, 26, 27, 28, 29

***CTC* – Continuous Time Conflict**: Defines a time interval in which two agents would collide if they move over the same node or edge. Related to the PSIPP's algorithm. 22, 23, 24, 89

***DES* – Discrete Event Simulation**: A simulation where time progression is modeled explicitly, through the execution order of discrete events. 69, 70, 71, 72

***EECBS* – Explicit Estimation CBS**: A bounded-suboptimal variant of CBS that uses online learning to obtain inadmissible estimates. 24

***FCS* – Fleet Control System**: The central warehouse agv fleet control management system currently used at KIN. iv, viii, ix, 1, 2, 3, 40, 41, 42, 43, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 57, 61, 64, 65, 66, 67, 68, 69, 72, 73, 75, 81, 82, 83, 84, 87, 89

***FCS Fake* – Fleet Control System - Fake**: A simplified simulation environment that closely mimics the FCS system. 3, 53, 54, 55, 57, 69, 70, 71, 72, 73, 77, 81

***IM* – Interval Manager**: The planning component responsible for managing graph-entity safe intervals. 58, 59, 60, 61, 62

***KIN* – Knapp Industry Solutions**. 1, 40

Lanemap: The shared road network that agents navigate on in FCS. iv, 40, 41, 42, 43, 44, 47, 48, 50, 51, 57, 62, 72, 73, 75, 76, 77, 82, 84, 87, 91

***MAPD* – Multi-Agent Pickup and Delivery**: Extends the (MAPF) framework to include dynamic and asynchronous tasks of transporting items between specified pickup and delivery locations. Agents continuously adapt to new tasks ensuring collision-free navigation throughout their operations. iv, viii, ix, 1, 2, 3, 4, 30, 31, 32, 37, 39, 52, 53, 56, 69, 89, 90, 91

***MAPF* – Multi-Agent Path Finding**: The problem of finding a collision free path for a set of agents from their start to their goal locations. iv, viii, ix, 1, 2, 4, 5, 7, 8, 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, 25, 26, 27, 29, 30, 31, 32, 35, 36, 37, 38, 52, 53, 56, 57, 58, 60, 63, 64, 65, 68, 80, 81, 86, 87, 89, 91

MAPF-POST: A robust plan execution framework that post-processes a MAPF plan into a temporal network. 32

NetworkX : A Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks 72

NP-hard – **Non-deterministic Polynomial-time hard**: The set of combinatorial problems for which no polynomial-time algorithm is known. 13, 18

OM – **Order Manager**: The FCS component responsible for distributing orders and managing their life-cycle. 52, 54, 55, 56, 70

PP – **Prioritized Planning**: A MAPF approach where agents are assigned fixed priorities, and each agent plans its path sequentially while considering higher-priority agents as obstacles. 19, 20, 21, 22, 32, 61, 77, 85, 89, 90

Protobuf: A language-neutral, platform-neutral extensible mechanisms for serializing structured data 72, 73

PSIPP – **Prioritized Safe Interval Path Planning**: A continuous time, euclidean roadmap extension of the classical PP algorithm. PSIPP leverages SIPP for underlying path planning together with pre-computed continous time conflicts for efficient collision avoidance on 2D roadmaps. 22, 23, 32, 57

PyCharm: Fully features (Python) IDE by © JetBrains. 69

Pygame: A set of Python modules designed for writing video games. 71

Python: An interpreted, high-level and general-purpose programming language. 69, 71, 72

RHCR – **Rolling Horizon Collision Resolution**: A dynamic horizon window framework that allows to solve MAPD problems as a series of sequential windowed MAPF instances. viii, 31, 37, 38, 39, 58, 60, 86, 87

RHCRE – **Rolling Horizon Collision-Resolution and Execution**: A robust real-world Multi-Agent Pickup and Delivery (MAPD) framework that is central to this thesis. iv, viii, 4, 31, 33, 38, 39, 86, 89

SAPF – **Single-Agent Path Finding**: The problem of finding a collision-free path for a single agent from a start to a goal location. 4, 7, 8, 91

Simpy: A process-based discrete-event simulation framework based on standard Python. 70

SIPP – **Safe Interval Path Planning**: A single-agent path planning algorithm that works with continous time. It avoids dynamic object collisions by leveraging safe-interval comparison strategies. viii, 4, 7, 8, 9, 10, 12, 23, 24, 27, 28, 57, 58, 60, 61, 80

SIPP-FCS – **Safe Interval Path Planning - FCS**: A Safe Interval Path Planning adaption for orientation based clothoid curve roadmaps. 61, 63

SM – **Shuttle Manager**: The central controller responsible for managing planning and order and task distribution between agents and the order manager. viii, 52, 54, 55, 56, 57, 65, 70

SSI – **Safe Start Intervals**: A method to prevent agents from moving over the start locations of other agents. viii, 21, 22, 57, 58, 60

Space-Time A*: A pathfinding algorithm that extends the A* algorithm by incorporating time as a third dimension in the search space. Space-Time A* is used to plan paths in dynamic environments 8, 19

TOPPRA – **Time-Optimal Path Parameterization**: Method to compute time-optimal path parametrization for robots subject to kinematic and dynamic constraints. 61, 73, 74

VRP – **Vehicle Routing Problem**: The problem of finding the optimal set of routes for a fleet of vehicles to deliver to a given set of customers. 7