Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Cybernetics

Bachelor's Thesis

2020 Filip Rýzner

Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Cybernetics

Degree of Study: Open Informatics

Multi-agent Path-finding for trains with breakdowns

Document purpose: Bachelor's Thesis
Author: Filip Rýzner
Supervisor: Ing. Martin Schaefer
Date of Submission: May 2020

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Rýzner Filip**	Personal ID number:	**467634**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Branch of study:	**Computer and Information Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Multi-Agent Pathfinding for Trains with Breakdowns

Bachelor’s thesis title in Czech:

Multi-agentní plánování tras vlaků s poruchami

Guidelines:

We consider a simplified model of the train infrastructure. The task is to find collision-free paths for trains with given origin and destination. The optimization criteria might be to minimize makespan of the plan or maximize the number of trains to reach the destination in a given time limit. The trains move at various speeds and suffer from malfunction. The problem is inspired by the Flatland Challenge.

1. Study the related literature on the most related multi-agent pathfinding problems.
2. Formulate the problem and propose a heuristic algorithm to solve the problem.
3. Evaluate the solution quality and the computational performance in simulation on various Flatland scenarios.

Bibliography / sources:

[1] Stern, Roni, et al. "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks.", 2019

[2] Barták, Roman, Jiří Švancara, and Marek Vlk, Scheduling Models for Multi-Agent Path Finding ,2017

[3] Čáp, Michal, et al. "Prioritized planning algorithms for trajectory coordination of multiple mobile robots." IEEE transactions on automation science and engineering, 2015

[4] Felner, Ariel, et al. "Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges." Tenth Annual Symposium on Combinatorial Search, 2017

Name and workplace of bachelor’s thesis supervisor:

Ing. Martin Schaefer, Artificial Intelligence Center, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **12.01.2020** Deadline for bachelor thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

Ing. Martin Schaefer Supervisor’s signature	doc. Ing. Tomáš Svoboda, Ph.D. Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

Prohlášení autora práce:

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne .... Podpis autora práce ....

Author statement for undergraduate thesis:

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date .... Signature ....

Acknowledgement

I would like to thank Ing.Martin Schaefer for his patient guidance of my work, for always being an inspirational supervisor, and for introducing me to such an interesting topic. Furthermore, I am grateful to my whole family for their continuous support during my studies.

Abstrakt

V této práci představíme a formalizujeme problém multi-agentního hledání cest pro vlaky (MAPF-T), což je zobecnění MAPF, kde agenti trpí poruchami imobilizujícími je po náhodný počet časových kroků, mají různé rychlosti pohybu a zároveň definujeme maximální počet časových kroků pro vyřešení instance problému. V naší práci představíme p-TPG, dvoustupňový přístup k řešení MAPF-T, který v první fázi využívá prioritní plánování pro hledání nekonfliktních cest. V druhé fázi p-TPG sestavuje graf časového plánu (temporal plan graph - TPG), který při exekuci plánů slouží k prevenci konfliktů, když dojde k poruchám agentů. Spolu s p-TPG představíme čtyři nové heuristiky pro určení priorit agentů, které zlepšují výkonnost prioritního plánování. Dále představíme úpravu konstrukčního algoritmu TPG, která snižuje dobu běhu algoritmu až 25-krát. Nakonec ukážeme, že pořadí priorit agentů, která upřednostňují agenty s delší dobou vykonání cesty, dominují ostatní prezentovaná řazení agentů ve všech prezentovaných instancích MAPF-T.

Klíčová slova Multi-Agentní plánování, Nedokonalé vykonávání plánů, Agenti s poruchami, Plánování agentů s různými rychlostmi, Umělá inteligence

E-mail autora f.ryzner@gmail.com
E-mail vedoucího práce martin.schaefer@fel.cvut.cz

Abstract

In this work, we introduce and formalise the problem of multi-agent path-finding for trains (MAPF-T), which is a generalisation of the multi-agent path-finding (MAPF), where agents suffer from breakdowns immobilising them for a random number of time-steps, have different speeds of movement and we define a time-step deadline for solving each instance of the problem. We present the p-TPG, a two-stage approach for solving the MAPF-T, which in the first stage uses the prioritised planning to plan non-conflicting agent paths. In the second phase, the p-TPG constructs the temporal plan graph (TPG), which then serves to prevent conflicts when agent breakdowns occur during plan execution. Alongside the p-TPG, we introduce four new agent ordering heuristics that improve the performance of the prioritised planning. Furthermore, we introduce a complexity reducing improvement for the TPG construction algorithm, which decreases the run-time of the algorithm up to 25-times. Lastly, we demonstrate that the agent priority orderings that prioritise agents with longer path execution times dominate all other presented orderings on all presented MAPF-T instances.

Keywords Multi-Agent Path-Finding, Imperfect plan execution, Agents with breakdowns, Agent planning with speeds, Artificial Intelligence

Author's e-mail f.ryzner@gmail.com
Supervisor's e-mail martin.schaefer@fel.cvut.cz

Contents

List of Figures

List of Tables

Acronyms

A*	A* search algorithm
AI	Artificial Intelligence
ASP	Answer Set Programming
BFS	Breadth-first search
CAT	Conflict Avoidance Table
CBS	Conflict Based Search
CT	Conflict Tree
DBS	Death Based Search
EPEA*	Enhanced Partial Expansion A*
Flatland	Flatland Challenge simulation
HCA*	Hierarchical Cooperative A*
LRA*	Local Repair A*
ID	Independence detection
I-kR-CBS	Improved k-robust Conflict Based Search
ICBS	Improved Conflict Based Search
ICT	Increasing cost tree Search
ILP	Integer Linear Programming
kR-CBS	k-robust Conflict Based Search
MA-CBS	Meta-Agent Conflict Based Search
MA-DBS	Meta-Agent Death Based Search
MAPF	Multi-agent path-finding
MAPF-B	Multi-agent path-finding with breakdowns
MS	Make-span
OD	Operator decomposition
PEA*	Partial Expansion A*
SAPF	Single agent path finding
SAT	Boolean satisfiability problem
SBB	Schweizerische Bundesbahnen
SIC	Sum of Individual Costs
TPG	Temporal plan graph
WHCA*	Windowed Hierarchical Cooperative A*

Chapter 1

Introduction

Train routing and scheduling are one of the most crucial components ensuring an efficient railway operation. Given the structural and engineering requirements for railway extension, the railway capacity cannot so simply be increased by expanding the existing infrastructure without a substantial investment. Therefore, railway operators regularly aim to boost the existing railway capacity by sophisticated planning and optimisation of train routes and schedules. This problem is especially relevant in countries that have a hilly or otherwise diverse landscape, which makes railway construction even more complicated.

Switzerland is a textbook example of a country where elaborate engineering efforts are required to construct railways as mountain crests cover a large part of the country. Therefore, to gather learning for real-world planning models, the Schweizerische Bundesbahnen (SBB) developed a grid world model that simulates the dynamics of train traffic on simplified railway infrastructure. This model was made available to the public through an international competition named "The Flatland Challenge" (SBB, AI Crowd [2020]), which aimed to address SBB's train scheduling problem on the model. The competition encouraged participants to employ reinforcement learning in order to find their solutions; however, we believe that we can obtain competitive results by utilising well-researched approaches for multi-agent path-finding problems (MAPF).

MAPF is the problem of finding non-conflicting paths for all agents respecting their starting and target locations in a given environment Ma *et al.* [2017]. With multiple agents present in the problem, it is crucial to guarantee that in addition to planning agents' paths in accordance with a defined objective function, all agents will be able to follow their paths concurrently and without any collisions. MAPF is a well-researched field in artificial intelligence (AI), robotics, theoretical computer science as well as operations research (Ma *et al.* [2017]). Interestingly, MAPF algorithms can be used to solve many real-world problems including planning movement for robots in warehouses (Hönig *et al.* [2019]), solving aircraft towing challenges (Morris *et al.* [2016]), navigating characters in video games (Silver [2005], Stout [1998]) or planning routes for office robots (Veloso *et al.* [2015]).

In our work, we will utilise the model provided by SBB for the Flatland Challenge; thus, we will focus on railway usage optimisation. Although the grid-based model with discrete time introduces a substantial simplification compared to the real world, it still has trains with different speeds and simulates random occurrence of train breakdowns. Different speeds of trains increase the complexity of the planning problem, but the randomly occurring train breakages create a serious issue as they cannot be accounted for during the path planning process itself. Therefore, we will not only have to develop a reliable planning algorithm, but also a reliable path and computationally efficient plan execution algorithm that will ensure that trains finish their routes without colliding with each other even when breakdowns occur.

Our goal is to maximise the number of agents who complete their paths in the simulation given the time-step deadline defined for each instance. This was the main criterion for participants in the "Flatland Challenge", and we note that no submission reached the completion rate of 100%. Furthermore, our secondary goal will be to minimise the sum-of-individual costs across trains in our simulation.

Our work starts with a problem definition in Chapter 2, where we provide an overview of the Flatland model and we introduce a mathematical formulation of it. We then present a literature review in Chapter 3, which provides a categorised overview of significant advancements in multi-agent path-finding approaches over the last 40 years. Moreover, in Chapter 4, we present a detailed description of the p-TPG approach that we will use for solving instances of the Flatland Challenge. In Chapter 5, we describe the test instances used for the testing of p-TPG and present obtained results. Finally, in Chapter 6, we summarise our findings.

Chapter 2

Problem Definition

In this chapter, we will first describe the properties and objectives of the Flatland Challenge simulation (Flatland) provided by the SBB on the AI Crowd web (SBB, AI Crowd [2020]). Then we will introduce a mathematical formulation of our planning problem in accordance with the terminology introduced in Stern *et al.* [2019].

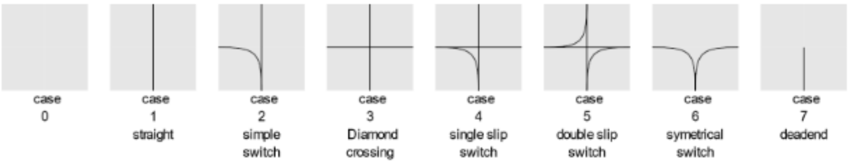
2.1 Flatland introduction

The Flatland is a discrete-time grid-based simulation, where we have to plan paths for trains in a way that as many as possible reach their destinations in a specific time-step limit determined by the simulation. Therefore, it is a case of MAPF, where the grid allows movement in 4 directions and agents in the problem correspond to trains in the simulation.

2.1.1 Flatland grid

The Flatland 4-way grid is composed of N rows and M columns and agents that move from their given starting grid position to their target position. Each tile in the grid is of 1×1 size and contains one of 7 base types of rails, which are depicted in Figure 2.1. Furthermore, each of these base rail types can be rotated by either 0, 90, 180 or 270 degrees or reflected along the vertical or horizontal axis. This results in 27 types of rail tiles, accounting for the fact that some rotations and reflections are identical with the original tile.

Figure 2.1: Flatland base rail types



source: SBB, AI Crowd [2020]

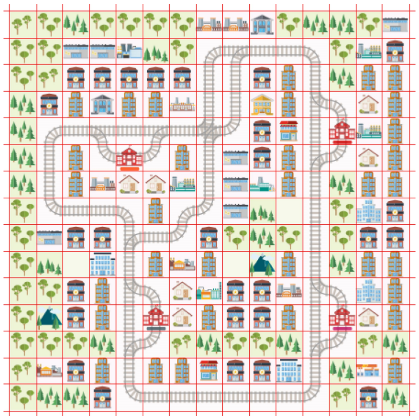
Contrary to standard grid environments, the railway representation imposes further constraints on agents' movement. Specifically, depending on the tile type, the agent can continue movement in only up to 2 directions upon entering a tile from a valid grid position. For example, after entering the tile with a straight railway depicted in case 1 (Figure 2.1) agent can only continue moving forward, while upon entering tile depicted in the case 5 (Figure 2.1) agent can continue moving forward or make left or right turn depending on the side in which it entered the tile. The direct consequence the railway representation is that agents cannot easily change their direction of movement. Specifically, an agent can only change its movement direction by 90 degrees after execution a left or right turn or by 180 degrees after it reaches a dead-end (case 7, Figure 2.1), where it can turn around and continue moving.

Furthermore, the Flatland grid contains only relatively sparse railways, as depicted in Figure 2.2. This, together with the limitation on movement directions imposed by tile types, increases the complexity of the MAPF problem as agents cannot easily change their direction of movement or avoid each other when travelling in opposite directions; thus we may have to plan long detours or waiting to avoid agents conflicts. Specifically, conflicts occur when at least two agents would occupy the same tile in one time-step or would directly exchange positions in one time-step.

2.1.2 Flatland agent

Each agent in the Flatland has a determined starting and target position. The starting position of an agent corresponds to a tile on the grid, and the agent

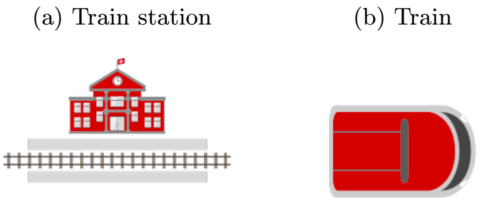
Figure 2.2: Flatland map before plan execution with grid illustration



source: SBB, AI Crowd [2020]

always starts with a valid orientation (a train will never be placed sideways on a railway segment); however, all agents start outside of the grid and have to be placed on it. We also note that several agents can have the same starting position; therefore, the planning must also determine the order and the moment of agent's placement on the grid. Finally, agent's target position always correspond to one of the stations on the grid (depicted in Figure 2.3) and, similarly as with starting positions, multiple agents may have the same target position. Furthermore, agents immediately disappear from the grid upon reaching their target.

Figure 2.3: Graphic representation of train stations and trains in Flatland



(a) Train station

(b) Train

source: SBB, AI Crowd [2020]

The logic of putting agents on grid further increases the complexity of the planning as there are $k!$ possible placement orderings for a tile where k agents start their movement, moreover, even if the ordering was determined, placements can happen at various time-steps, thus again creating many planning options.

Flatland agent movement speed

Agents in the Flatland can either move forward, execute a turn if possible or wait in their current position. In a standard setting, agents would be able to execute exactly one move during each time-step; however, the Flatland assumes agents with different speeds of movement. These speeds are randomly chosen from a pre-selected set of allowed speeds, where all of them have to belong to an interval of $(0, 1]$.

Given that all grid tiles are of 1×1 size, an agent with a speed value below 1 may require several time-steps to traverse one tile. During each turn, an agent accumulates a traversal fraction equal to its speed, and when it is equal or bigger than 1, it resets and the agent moves to a next tile. More precisely, the mechanism of agent's movement is illustrated in the Algorithm 1. Figure 2.4 shows the Flatland grid during the plan execution when agents are travelling to their targets.

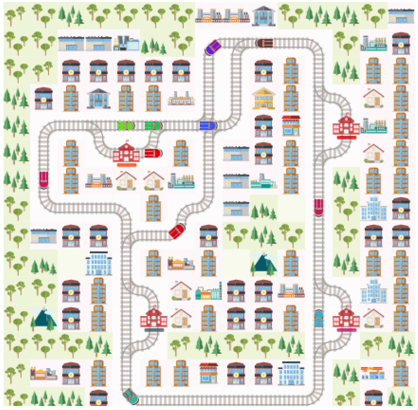
Algorithm 1 Flatland agent movement with different speeds

```
$agent\_speed \in (0, 1]$  //Agent has some speed
$traversal\_fraction \leftarrow 0$
while agent_not_done == True do
    if agent moving then  //Agent is traversing tile, not waiting
        $traversal\_fraction += agent\_speed$
    if $traversal\_fraction >= 1$ then
        $traversal\_fraction \leftarrow 0$
        execute_move()  //The agent moves
```

Flatland agent malfunction

To better simulate the real-world train behaviour, the Flatland agents Are subjects to breakdowns. All agents have the same non-zero probability that they suffer a breakdown of a duration randomly chosen from a pre-specified interval. Agent breakdown can occur at any time; therefore it can happen when the agent is moving across the tile, when waiting at a tile or when the agent is not even on the grid (agent can still be placed on the grid, but will be broken). When the malfunction occurs, the agent is immobilised for a given number of time-steps and cannot perform any movement. Furthermore, if the breakdown occurred when the

Figure 2.4: Flatland during plans execution



source: SBB, AI Crowd [2020]

agent was traversing a tile, its movement direction cannot be changed in any way after the break down ends; hence the agent will finish the precise traversal it started before the breakdown.

The existence of train breakdowns has significant consequences for the planning and execution of each Flatland instance. With random breakdowns occurring, it is no longer possible to plan all agents' movements before the Flatland instance is run, but we will also have to manage our plans during its execution to prevent conflicts from occurring as unexpected breakdowns can cause de-synchronisation of agents' plans.

2.1.3 Simulation objectives

Time-step limit

Each Flatland instance has a specific time-step limit assigned and all agents are required to reach their target positions in under the limit. An agent that cannot reach its target position under the time-step limit is put among uncompleted agents. The time-step limit for each instance is obtained as follows:

$$T_{max} = 8 * (grid\_width + grid\_height + \lceil \frac{|A|}{|C|} \rceil)$$

where the A is a set of agents in the Flatland instance, C represents the number of cities in the Flatland grid and $|.|$ is a function determining the cardinally of a set.

Objectives

Finally, the main objective of the Flatland Challenge was to maximise the number of agents who successfully reach their target position in under a given time-step limit set for each Flatland instance. Therefore, we will adopt the same primary objective. Furthermore, our secondary criterion will be the minimisation of the sum of individual cost, which represents the sum of all costs across all agents. All agents incur per time-step cost equal to their speed from the start of the Flatland instance until they reach their target location. Finally, we will also analyse the make-span, which represents the total time until the last agent reaches its destination (Barták *et al.* [2018]).

2.2 Mathematical formulation

In this section, we will introduce a mathematical formulation of the Flatland MAPF problem introduced in the previous section. We will base the formulation on the outline introduced in Stern *et al.* [2019]; however, we will modify and extend it in order to include the agent agents with different speeds, agent breakdowns, and other specific issues related to our problem. From this point onward, we will refer to our problem as the Multi-agent path-finding for trains (MAPF-T) to differentiate it from the standard MAPF.

An instance of the MAPF-T problem with K agents can be described as a 10-tuple in the following way:

$$\Gamma = \langle G, A, \lambda_S, \lambda_F, H, v, \gamma, \delta, P, T_{max} \rangle,$$

where:

- G = (V, E, ϵ) is a weighted directed graph
- A = $\{a_0, \ldots, a_{k-1}\}$ is an ordered finite set of agents
- $\lambda_S : A \to V$ is an injective function setting agent's start position
- $\lambda_F : A \to V$ is an injective function setting agent's target position
- H = {Wait, Forward, Left, Right} is the ordered set representing possible agent's actions/moves

- $\upsilon : (V, H) \to V$ is an injective agent movement function
- $\gamma : A \to (0, 1]$ is a surjective function determining agent's speed
- $\delta : A \to [0, \mathbb{R}]$ is a surjective function setting agent's cumulative speed
- $P : A \to \mathbb{N}$ is random process function determining train malfunctions
- T_{max} represents the maximum allowed make-span of the solution

2.2.1 Graph representation of Flatland grid

As defined above $G = (V, E, \epsilon)$ is a weighted directed graph, where

$$\epsilon(e) = (u, v) \quad \forall u, v \in V, e \in E$$

is an edge starting in vertex u and ending in vertex v.

To be able to create an accurate graph representation of the $N \times M$ tile sized Flatland grid, we first introduce the following ordered agent orientations set and an agent orientation determining function:

- O = {N, E, S, W} is an ordered set representing orientations agent
- $\xi : A \to O$ is the surjective function setting agent orientation

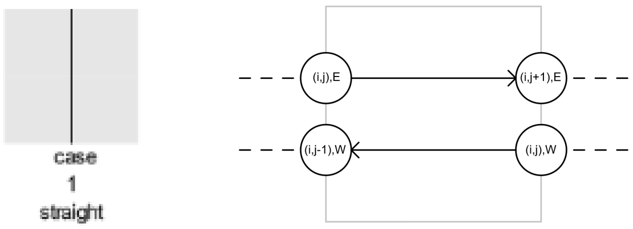
Next, we define a vertex in our graph representation as a 3-tuple:

$$v = ((i, j), o) \quad i \in N, j \in M, o \in O$$

where the (i, j) corresponds to coordinates of a tile in the Flatland grid and o represents one of the valid orientation the agent can have upon entering the corresponding tile. The inclusion of agent's orientation is necessary to accurately represent the railway logic introduced in the previous section.

To better illustrate the logic of the graph representation, we present two examples in Figures 2.5 and 2.6. The Figure 2.5 demonstrates how a straight rail tile (case 1, Figure 2.1) at position (i, j) is converted to our graph representation. For example, if an agent arrives to a tile (i, j) and is facing East (orientation E), its location will correspond to the vertex $((i, j), E)$ in the graph representation. The agent can either wait in its position, which would correspond to the edge

Figure 2.5: Graph representation example - case 1 railway

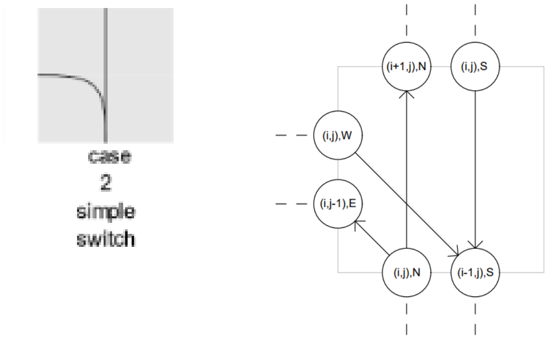


(a) Case 1 railway

(b) Graph representation

$[((i,j),E),((i,j),E)]$ or it can move forward to the next tile, which corresponds to the edge $[((i,j),E),((i,j+1),E)]$. The Figure 2.6 presents an additional example of the logic, but for a slightly more complex rail tile (case 2, Figure 2.1).

Figure 2.6: Graph representation example - case 2 railway



(a) Case 2 railway

(b) Graph representation

2.2.2 Agent formulation

Each agent has a starting and target position determined by functions λ_S and λ_F respectively and a fixed speed determined by the function γ. We define, the injective function determining agent's position at the time-step t as follows:

$$\lambda_t : A \to V$$

Furthermore, the execution priority of agents in the Flatland is determined using the priority function p defined as follows:

$$p : A \to \mathbb{N}_0$$

$$\text{where:} \quad p(a_i) > p(a_j) \quad \text{for:} \quad i < j$$

Contrary to standard multi-agent engines, where state validity is evaluated after all agents are moved, Flatland executes agent's actions one by one during every time-step. This can lead to additional conflicting situations. For example, given two agents where one aims to enter a position, which the other agent is supposed to leave in the time-step t, a conflict would arise if the entering agent had a higher priority. This is because the Flatland simulator would try to first execute the action of the higher priority agent, but that would not be valid as the lower priority agent would not have left the position in question yet.

Agent action plans

An agent has to perform an action during every time-step until it reaches its target position. Each agent has to either wait or move forward, turn left or turn right. Agent's movement is represented by the function v, which we defined above.

To be able to correctly define agent's move, we need to define its duration, which is determined by the agent's speed as follows:

$$\rho = \left\lceil \frac{1.0}{\gamma(a_i)} \right\rceil \qquad \gamma(a_i) \in (0, 1] \tag{2.1}$$

where a_i represents the $i - th$ agent in the Flatland. Subsequently, we define a move by the i-th agent as:

$$v(\lambda_t(a_i), h) = \lambda_{t+\rho}(a_i) \quad \text{where} \;\; [\lambda_t(a_i), \lambda_{t+\rho}(a_i)] \in E, \; h \in H$$

which says that if an agent a_i executes the action h at the vertex $\lambda_t(a_i)$, it will move to the vertex $\lambda_{t+1}(a_i)$ over the next ρ time-steps.

Next, we define a sequence of n actions planned for the i-th agent as:

$$\pi_{a_i} = (h_1, \dots h_n) \quad \forall i, \; h_i \in H$$

Moreover, we say that $\pi_{a_i}[t]$ denotes the location of the agent a_i after t time-steps. Formally, we define the $\pi_{a_i}[t]$ as follows:

$$\pi_{a_i}[t] = v(v(\dots v(\lambda_S(a_i), h_0), h_{t-1}), h_t)$$

Finally, we say that a sequence of actions for an agent a_i is a single-agent plan if starting executing actions from the sequence at $\lambda_S(a_i)$ leads to $\lambda_F(a_i)$ in $|\pi|$ steps (Stern *et al.* [2019]). A solution of the MAPF-T is a set of single-agent plans for all agents, which do not contain collisions and its length is shorter than a specified limit.

Agent speed

Every agent has a different speed in a range from $(0, 1]$, which is determined by the function γ defined at the beginning of this section. Moreover, if the agent is moving its cumulative speed determined by the function δ increases by his speed determined by γ in every time-step and agent can only move from one tile to another when its cumulative speed is $\delta(a_i) \geq 1$. We described this logic in the Algorithm 1 located in the previous section and defined the number of steps the agent requires to traverse a tile in the Equation 2.1.

To sum the movement with speed, we include a final simple example: if agent's speed is equal to 1 it can move from a vertex $v1$ to its adjacent vertex $v2$ in every turn, however if the agent's speed is equal to 0.3 it only can change its position after 4 time-steps, when his cumulative speed $\delta(a_i) = 1.32$ thus $\delta(a_i) \geq 1$. Finally, we again highlight that in the previous example, the surplus accumulation of 0.32 would not carry over to the next time-step.

Agent breakdowns

Flatland agent's can breakdown during every turn, thus can even be placed on the grid already broken. All agents have the same probability p_{break} of breaking down in every turn, which can be set differently for every instance. Furthermore, each instance has a set break-down duration interval $[min_{break}, max_{break}]$. The mechanism of breakdowns is relatively simple, and we briefly summarise it in the Algorithm 2.

Broken agents remain inactive for the next several time-steps based on the *duration* of the breakdown.This can be formulated as:

$$\lambda_t(a_i) = \lambda_{t+1}(a_i) = \ldots = \lambda_{t+duration}(a_i)$$

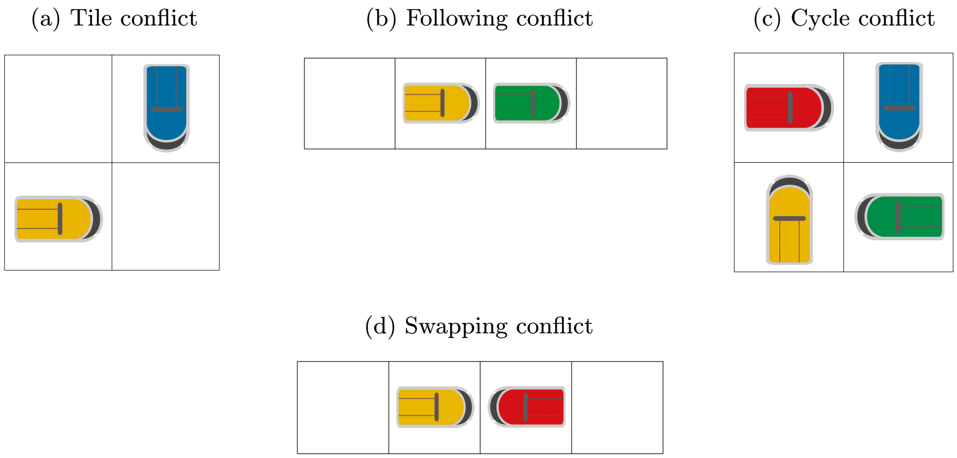
Agent conflict types

In this section, we define conflicts that can arise between agents during the execution of their plans. It is crucial to understand that a set of single-agent plans represents a valid solution to the MAPF-T problem if and only if it does not contain any conflicts.

Algorithm 2 Flatland breakdowns mechanism

```
$p_{break} \leftarrow [0,1]$
$min_{break} \leftarrow x \geq 0$
$max_{break} \leftarrow y \geq min_{break}$
while $simulation\_running$ do
    for $agent \in active\_agents$ do    //agents not yet in target position
        if agent_not_broken then
            $x$ = random_float(0,1)    //random float in [0,1]
            if $x \leq p_{break}$ then
                $duration$ = random_int($min_{break}, max_{break}$)
                assign_breakdown($agent, duration$)    //break agent
```

Figure 2.7: Illustration of conflict situations



(a) Tile conflict

(b) Following conflict

(c) Cycle conflict

(d) Swapping conflict

source: SBB, AI Crowd [2020]

The definition of our conflict may be slightly confusing as we cannot determine conflicts based on single graph representation vertex or edge. We have to account for the fact that a Flatland grid tile, where conflicts occur, can be represented by up to 8 vertices in the graph representation.

Therefore, before we define the possible conflict situations, we note that from now on, we will use the $*$ symbol in the vertex notation $v = ((i,j), *)$ to represent arbitrary valid agent's orientation. In order to correctly detect conflict on the level of Flatland grid tiles, the agent orientation will not be taken into consideration when comparing agents position vertices. Hence the conflict detection will be based solely on the corresponding tile coordinates represented by (i,j).

We will now list and describe all conflict situations that can arise in our MAPF-T problem alongside with their visual depiction in Figure 2.7. Let a_i and a_j be a two agents with single-agent paths π_{a_i} and π_{a_j} respectively.

A) **Tile Conflict**

A tile conflict between two agents occurs when both agents plan to enter a vertex related to the same tile at the same time step, hence the graph vertex might be different, but the conflict is caused by the underlying relationship to the same tile (Figure 2.7a).

Formally, we define the tile conflict as follows:

$$\pi_{a_i}[t] = \lambda_t(a_i) = ((i,j),*) = \lambda_t(a_j) = \pi_{a_j}[t]$$

B) **Following Conflict**

A following conflict between two agents occurs when an agent with a higher priority at time-step t is planning to move to a tile that was occupied by an agent with lower priority at time t-1 (Figure 2.7b). As explained previously in this chapter, the following conflict is implied by the way Flatland processes agent actions during each time-step.

Formally, we define the following conflict as:

$$\pi_{a_i}[t] = \lambda_t(a_i) = ((i,j),*) = ((i,j),*) = \lambda_{t-1}(a_j) = \pi_{a_j}[t-1]$$
$$\textit{where:}\;\; p(\lambda_j) > p(\lambda_j)$$

C) **Cycle Conflict**

A cycle conflict occurs between a set of agents occurs when a rotation circle is formed (Figure 2.7c).

Formally, we describe the cycle conflict as follows:

$$\pi_{a_1}[t] = \lambda_t(a_0) = ((i_0,j_0),*) = ((i_0,j_0),*) = \lambda_{t-1}(a_1) = \pi_{a_1}[t-1]$$
$$\pi_{a_1}[t] = \lambda_t(a_1) = ((i_1,j_1),*) = ((i_1,j_1),*) = \lambda_{t-1}(a_2) = \pi_{a_2}[t-1]$$
$$.$$
$$.$$
$$.$$
$$\pi_{a_k}[t] = \lambda_t(a_k) = ((i_n,j_n),*) = ((i_n,j_n),*) = \lambda_{t-1}(a_0) = \pi_{a_0}[t-1]$$

The cycle conflict is independent of agents' priorities. Therefore, we needed to formalise it as it the logic differs from the following conflict, although they are logically related.

D) **Swapping Conflict**

The swapping conflict occurs if and only if the two agents plan to swap positions, more precisely tiles, in one times step (Figure 2.7d).

Formally, we define the swapping conflict as follows:

$$\pi_{a_i}[t-1] = \lambda_{t-1}(a_i) = ((i,j),*) = ((i,j),*) = \lambda_t(a_j) = \pi_{a_j}[t]$$

$$and$$

$$\pi_{a_i}[t] = \lambda_t(a_i) = ((i,j),*) = ((i,j),*) = \lambda_{t-1}(a_j) = \pi_{a_j}[t-1]$$

We note that the swapping conflict is technically a case of the Cycle conflict for two agents, but we list it to increase the clarity of the description.

The set of non-conflicting plans that are shorter than T_{max} represent a potential solution to our problem.

2.2.3 Solution criteria

Keeping the aforementioned definition of the solution in mind, we will formalise the conditions that we will use for the selection of the best solution.

The ideal solution of a Flatland instance maximises the number of agents who reach their target position in under T_{max} time-steps. Moreover, our secondary criterion is to minimise the sum of individual costs (SIC). Therefore, we define the ideal solution to a k-agent MAPF-T problem as follows:

$$\min SIC(\underset{\pi_i}{\text{argmax}}(|A_C|))$$

where A_C denotes the set of agents who have successfully completed their paths, SIC denotes the sum of individual costs of the solution π_i and the function $|.|$ determines the size of the set. Alternatively, the completion rate defined as $\frac{|A_C|}{|A|}$, where A represents the set of all agents, can be maximised instead of the absolute number of agents reaching their target.

Chapter 3

Literature Review

In this chapter, we will present a literature review highlighting significant advancements relevant to the path-finding problems. We will first provide a brief overview of pertinent algorithms for the Single-Agent path-finding (SAPF) and explain why these algorithms may not be, without further adjustments, used to solve the Multi-Agent path (MAPF) finding problems. Next, we will discuss the beginnings of the MAPF algorithms and outline general settings for MAPF. Furthermore, we will review and classify state-of-the-art algorithms into optimal and sub-optimal while focusing predominantly on reviewing the search-based MAPF approaches. Lastly, we will review the current advances in the MAPF field concerned with handling unexpected delays during the execution of agents' paths, a problem very similar to ours.

3.1 From Single-Agent to Multi-Agent Path Finding

The simplest algorithms used for path-finding are the Bread First Search (BFS) algorithm, which guarantees the optimality of the solution on an unweighted graph and the Dijkstra algorithm derived by Edsger W. Dijkstra (Dijkstra [1959]). However, the A* algorithm proposed by Peter Hart in Hart *et al.* [1968] is probably one of the most popular and effective algorithms used for SAPF. The algorithm considerably increased the search efficiency by utilising a heuristics function, which, if admissible, guarantees that the algorithm finds the optimal solution to the SAPF problem. Another interesting property of the algorithm is that when a node is expanded, and the heuristic is consistent, the optimal distance from a start node to that node is determined Nilson [1980]. However, should the heuristic function be improperly selected, then the A* algorithm can substantially under-perform Dijkstra's algorithm in terms of the search efficiency. In general, in their original form, SAPF algorithms are only suitable for solving the SAPF problem as they do not take into account the possibility of collisions, which arise when we plan routes for multiple agents in one environment.

The beginnings of MAPF algorithms can be traced back to the Erdmann & Lozano-Pérez [1987] who investigated how to prevent collisions when navigating multiple agents in a shared space. Erdmann & Lozano-Pérez [1987] focused on planning translations of planar objects and rotating planar articulated arms while assuming planning order determined by fixed priorities assigned to agents. The publication introduced decoupled and distributed approaches leading to independence or weak dependency of agents' planning problems. This lay the foundations for some future approaches relying heavily on agent priorities and reducing dependency between agents' planning problems.

In the next section, we will focus solely on approaches solving the MAPF. We will first introduce shared characteristics of the MAPF problem and then describe the core MAPF approaches and introduce a suitable categorisation of these approaches, ensuring a clearer arrangement of the overview.

3.2 Categorisation of MAPF approaches

In general, agents in a MAPF problem can operate in either distributed or centralised settings. In the centralised setting, we assume that a single decision-making authority fully controls the actions of all agents, whereas, in the centralised setting, we assume that all agents choose their actions themselves. Therefore, we can further categorise the distributed setting by agents' decisions protocols into cooperative and self-interested (Bnaya *et al.* [2013]). Cooperating agents will take actions in accordance with the global objective function, thus behave as if a single-decision making authority directed them, whereas self-interested agents require a coercive mechanism to adhere to the global objective function; otherwise, they will only try to maximise their own (Bnaya *et al.* [2013]).

A common objective of MAPF problems is to minimise the global cumulative cost. Therefore, the sum-of-individual-cost (SIC), defined as the sum of cost across all agents required to reach their destination and never leave it, is a popular choice in the MAPF research (Sharon *et al.* [2015a]). The total time until the last agent reaches its destination or so-called make-span (Barták *et al.* [2018]) is another common choice for the objective function. Finally, the selection of the objective function also depends on the goals of the research, thus can vary for specifically focused publications. One such example is the Ma *et al.* [2018], where the number of agents reaching their destination before a set time-step deadline is maximised.

One of the most distinctive characteristics of MAPF approaches is their optimality. The MAPF problem was shown to be NP-hard (Surynek [2010], Yu & LaValle [2013]) because the state-space grows exponentially with the number of agents in the problem. Therefore, some approaches sacrifice optimality for a more feasible run-time. We will now provide an overview of both optimal and sub-optimal approaches for solving the MAPF. However, we note that the overview is not meant to be completely exhaustive and that we will dedicate more space to search-based and tree-based solvers as they are the most relevant for our work.

3.2.1 Sub-optimal MAPF approaches

Sub-optimal approaches aim to quickly find paths for agents, especially when the number of agents in the problem is high and the optimal solution cannot be determined. However, it can occur that sub-optimal approaches do not even guarantee completeness (Sharon *et al.* [2015a]). We can generally classify sub-optimal approaches into three categories, as follows:

Search-based approaches

Some of the earliest sub-optimal search based algorithms were introduced by Zelinsky [1992] and later summarised with robustness improvement proposals such as search space transformation or node exploration limitations by Stout [1998]. Algorithms from this family[1] first plan routes for all agents independently and resolve conflicts only when they become imminent during the next step of the plan exe-

[1] later labelled as the ”Local Repair A*” (LRA*) by Silver [2005]

cution. Conflicts are resolved by searching for alternative routes from the agent's current location to the target destination while prohibiting the edge or vertex that is causing the conflict in the next step.

Well-known sub-optimal search-based approaches were introduced by Silver [2005] as a direct improvement on the Local Repair A* algorithms. According to Silver [2005] the Local Repair A* algorithms were often used in games at that time but suffered from deadlocks and strange behaviour in situations with bottlenecks. The most prominent algorithms introduced in Silver [2005] were the Hierarchical Cooperative A* (HCA*) and the Windowed Hierarchical Cooperative A* (WHCA*). Both algorithms made use of a conflict avoidance table tracking agents' routes in space-time to prevent agents collisions.

In the HCA* agents plan their routes sequentially in an order determined by their priorities. After an agent plans its route, it gets noted in the reservation table and agents with lower priority then treat it as an obstacle that needs to be avoided when they are planning their routes.

WHCA* was introduced as a modification of the HCA* to enable its usage in real-time applications such as video games. WHCA* only performs the non-conflicting planning in subsequent windows of a fixed size, where the conflict avoidance table is used to check for potential conflicts. Agent's paths outside of these windows are planned using SAPF approaches ignoring other agents. This approach guarantees that paths in windows remain without conflicts, while agents try to follow their best routes as determined by the SAPF planning. Furthermore, the segmentation of planning allows for changes in agents' priorities resulting in availability solutions, which under fixed-priority ordering could not be found. Both algorithms significantly outperformed the LRA* algorithm and currently remain a subject of research. Recent improvements include the introduction of the Conflict-Oriented Windowed Hierarchical Cooperative A* by Bnaya & Felner [2014], which improved the solution quality but is more time demanding compared to the baseline algorithm.

Lastly, Ma *et al.* [2019] reviewed the category of MAPF approaches based around priority planning, where agents are assigned fixed priority ordering and must ensure that their planned paths never interfere with the paths of agents with higher priority. In the publication Ma *et al.* [2019] also introduced a two-level search algorithm called Priority Based Search and modified the optimal Conflict Based Search introduced by Sharon *et al.* [2015a]. The resulting priority planning based approaches are not optimal nor complete but are highly efficient.

Rule-based approaches

Rule-based approaches are not at the core of the focus of this thesis, but we provide a brief overview of these approaches for completeness. These approaches impose specific movement rules, which affect how agents plan their paths and usually do not include massive search (Sharon *et al.* [2015a]). Although rule-based approaches are usually complete and have low computational cost, thus are fast at solving the problems, the quality of the solution tends to be low as it is sacrificed for the computational speed (Felner *et al.* [2017]).

The TASS algorithm by Khorshid *et al.* [2011] and Push and Swap algorithm by Luna & Bekris [2011] are the two commonly mentioned algorithms of this type. Push and Swap algorithm was enhanced several times resulting in the Parallel Push and Swap by Sajid *et al.* [2012] and Push and Rotate by De Wilde *et al.* [2014]. The TASS algorithm is known to be complete for tree graphs, while Push and Rotate is complete only for graphs where at minimum two vertices are not occupied by any agent at any time Sharon *et al.* [2015a]. Lastly, the BIBOX algorithm introduced by Surynek [2009] is complete for bi-connected graphs, while the enhanced version called diBOX developed by Botea & Surynek [2015] is complete even for strongly bi-connected directed graphs.

Hybrid approaches

We consider some approaches as hybrid because they execute a massive search as well as impose movement rules. Generally, these algorithms aim to prevent collisions and do not necessarily aim for the shortest paths. Furthermore, in the majority of cases, hybrid approaches are not complete and are better suited for planning in large open areas as they can suffer from deadlocks in maps that contain bottlenecks (Sharon *et al.* [2015a]).

One example of a hybrid approach was introduced by Ryan [2008]. The publication focuses on a centralised planning for MAPF and presents a new abstract representation fro MAPF, allowing for faster planning without sacrificing completeness. The main graph representing the environment is decomposed into sub-graphs while imposing entry and exit restrictions for each of them. The algorithm then solves for sub-graphs and computes transitions between them quickly and deterministically; thus performing a search in the space of sub-graph configurations.

Another example of a hybrid approach is the solver introduced by Wang & Botea [2011], which pre-computes the optimal paths for individual agents and also an alternative path between every two successive vertices an agent will be traversing according to its planned path. When a conflict is encountered the path of the conflicting agent is replanned using the pre-computed alternative paths. It is important to note that according to Wang & Botea [2011], the approach is complete only for grids with the slidable property, which are formally defined in their publication.

3.2.2 Optimal MAPF approaches

We will now review algorithms that solve the MAPF optimally while categorising them into three categories. As outlined before, solving the MAPF optimally is NP-hard and is the problem is very challenging, especially for complex instances that contain many agents with potentially highly conflicting paths.

Reduction based approaches

Reduction based approaches aim to reduce the MAPF problem to other problems that are already well studied in the field of computer science, such as SAT, Linear Programming or Constrained satisfaction problem Sharon *et al.* [2015b]. Although, these problems are often also NP-hard, high-quality approaches for solving them already exists. We note that many of these approaches were designed for the make-span objective function and cannot be easily modified to be used with the more common sum-of-cost objective function (Felner *et al.* [2017]). Furthermore, it should be noted that although optimal, these approaches are usually highly efficient only on small instances of the MAPF problems (Sharon *et al.* [2015a]).

One of the most popular reduction based approaches, according to Barták *et al.* [2018], was developed by Surynek [2012] and reduces the MAPF problem into the Boolean satisfiability problem (SAT). Other approaches include MinMakeSpan and MinTotalDist introduced by Yu & LaValle [2013], which model the MAPF problem as a network flow and then reduce it into the Integer Linear Programming problem (ILP) or the reduction to the Answer Set Programming (ASP) introduced in Erdem *et al.* [2013].

Search-based approaches

Search based approaches are popular in recent publications, and while they are often designed for the sum-of-cost objective function, they can as well be modified for the make-span objective function. The majority of search-based approaches is built around modified versions of the A* algorithm (Hart *et al.* [1968]). Moreover, several suitable MAPF heuristics were proposed across publications, including the standard Manhattan distance suitable for 4-way connected grid (Ryan [2008]), the sum of individual costs (Silver [2005], Standley [2010a]) as well as using forms of pattern-databases (Goldenberg *et al.* [2012], Goldenberg *et al.* [2014]).

In its elemental form, the A* operator represents actions that all k agents in the problem execute in a given time-step. However, the elemental form of the A* has several drawbacks when used for solving the MAPF. The first drawback is that the branching factor of the A* is exponential in the number of agents in the problem. For example, the case of a 4-way connected grid with the possibility of waiting corresponds to a branching factor of 5^k, where k is the number of agents. Another drawback is the potential size of the A* open-list. The open-list grows by up to b states every time a state from the list is expanded and its successors are added to it, where b denotes the number of successors of the expanded node.

Standley [2010a] criticised the greediness and sub-optimality of approaches introduced in by Silver [2005] and highlighted that higher-quality solution or alternative solutions cannot be obtained with these approaches even if we dedicated more computational time for calculations. As a follow-up on the previous research in the area of search-based approaches, Standley [2010a] introduced a first practical, admissible and complete search-based approach for solving the MAPF. The approach is based on two significant improvements of the elemental form of the A*, namely the operator decomposition and independence detection.

The operator decomposition (OD) decomposes the elemental A* MAPF operator into a series of operators, which leads to a significant reduction of the branching factor as the OD operator only considers one action corresponding to one agent at a time. Moreover, Standley [2010a] defines standard and intermediate search states, where the intermediate state occurs when at least one agent has already been assigned a move, but not all agents have been. After all agents have been assigned a move, a new standard state is created and all other intermediate states on the open-list that lead to its creation are ignored. This implies a significant reduction of nodes added to the open list. The branching factor of A* is reduced from b^k to k, but the dept of the goal node increases by the factor of k, where k is the number of agents in the problem. We note that the implemented search algorithm must allow agents to plan a move into spaces occupied by other agents with lower priority who are yet to be assigned move; otherwise optimality is not guaranteed (Standley [2010a]).

The independence detection (ID) framework reduces the branching factor of the A*. It sorts agents into independent groups and finds optimal solutions for each group of agents. We say that two groups of agents are independent if and only if the optimal solution for both groups can be found and they are not conflicting (Felner *et al.* [2017]). At first, each agent starts in its own group and a path is planned for him and noted into the conflict avoidance table. When a conflict between two groups, $g1$ and $g2$, is detected, the framework first tries to find an alternative non-conflicting path for $g1$ and vice versa. If that fails, the two groups are merged and paths for agents in the new group are planned together. This procedure is repeated until all independent groups of agents have been identified and optimal non-conflicting paths for all groups have been planned. It is worth noting that the run-time of the MAPF approach using the ID is dominated by the largest group of agents that corresponds to the largest independent sub-problem (Standley [2010a]). According to Felner *et al.* [2017] implementing the ID can result in up to an exponential speed-up obtained by the reduction of the number of agents in the planning problem.

The current state-of-the-art MAPF A* algorithm version provides a significant speed-up over the elemental algorithm, and it was first introduced as the Partial Expansion A* (PEA*) algorithm by Yoshizumi *et al.* [2000] to deal with the problem of generation of surplus nodes. When the PEA* expands a node n, up to b children are generated, but only those for whom it holds that $f(n) = f(b)$ are inserted into the open list. The other children are discarded and the node n is reinserted into the open list with the $f(b_{best})$ value corresponding to that of the best child. The f value function is defined as $f(n) = g(n) + h(n)$ and the $g(n)$ denotes the optimal cost of travelling from start to the node n and $h(n)$ is the heuristics estimation of the cost from the node n to the target position Goldenberg *et al.* [2012]. The algorithm was later improved to Enhanced Partial Expansion A* (EPEA*) by Felner *et al.* [2012] who introduced a way to only generate children of the node n for which it holds $f(n) = f(b)$. Later Goldenberg *et al.* [2012] and Goldenberg *et al.* [2014] applied the EPEA* to the MAPF and achieved significant improvements in run-times over all other optimal search-based approaches.

Tree based approaches

In the last category, we will review optimal approaches based on a tree data structure. These approaches are conceptually different from search-based approaches as they are not built around the A* algorithm.

The first prominent optimal tree-based approach was the increasing cost tree search algorithm (ICT), a two-level search algorithm with pruning rules, introduced in Sharon *et al.* [2013]. The ICT approach was up to 3 orders of magnitude faster than the A* based approaches on selected maps (Sharon *et al.* [2013]).

Another tree-based approach, which forms the core of the contemporary MAPF literature, is the Conflict Based Search approach (CBS). The CBS is a two-level algorithm introduced by Sharon *et al.* [2015a] as a complete and optimal solver able to solve large instances of MAPF problems with which the search algorithms would struggle given their branching factor.

The higher level of the CBS is the conflict tree (CT) which is a binary tree that guides the overall search for the solution. Each node of the tree contains constraints limiting paths for selected agents, agents' paths and cost of the solution based on the objective function. The root is an empty node with no constrains, and the solution is found at the node where all agents satisfy their constraints and there are no conflicts between their paths. On the lower level of the CBS is a fast single-agent path planning algorithm finding paths for individual agents subjecting to the restrictions imposed by the corresponding node in the conflict tree.

In case that a CT node contains conflicts, one of them is arbitrarily selected and resolved by splitting. During the splitting the CT node is split and two child nodes are created, where each of them prohibits one of the conflicting agents from entering the conflicting vertex. The CBS always constraints only two agents in each step even if there are more agents conflicting in the vertex in question.

In general, it was found that the CBS outperforms the state-of-the-art search algorithms in maps with bottlenecks, but struggles in open maps or MAPF problems where conflicts often occur as the conflict tree grows very fast. Therefore, the CBS in its elemental form is not suitable for a problem such as the Flatland because the sparse railway with a very high number of agents and the necessity to place agents on the grid lead to unacceptable planning times.

As the CBS arbitrarily chooses conflicts to resolve, it is very sensitive to these choices and if they are poor, they may significantly increase the size of the conflict tree (Felner *et al.* [2017]). Due to the space for improvement in the algorithm, CBS was recently a subject of several publications that aimed to improve its overall performance.

The Meta-agent CBS (MA-CBS) was the first improvement and was actually introduced alongside the CBS in the Sharon *et al.* [2015a]. The MA-CBS applies a conflict bound parameter, which determines after how many repeated conflicts are agents merged into a group and planned together at the low level. The conflict bound parameter can take values from 0 to $infinity$, where the value of 0 makes MA-CBS equivalent to the ID and as the value approaches $infinity$ the MA-CBS resembles the standard CBS.

Next, the Improved CBS (ICBS) defined by Boyarski *et al.* [2015] provided a significant speed-up compared to the elemental CBS by introducing conflict bypassing, fixed approach to conflict resolution based on categorisation by importance as well as the possibility to restart the construction of the conflict tree upon merging agents. A version of the CBS extended for a suitable heuristic was developed by Felner *et al.* [2018] and outperformed even the improved CBS by up to a factor of five (Felner *et al.* [2018]). Li *et al.* [2019a] recently introduced two additional admissible heuristics outperforming those introduced by Felner *et al.* [2018].

Finally, Li *et al.* [2019b] and Li *et al.* [2020] significantly boosted the performance of the CBS by significantly improving the resolution of corridor conflicts, rectangular conflicts and situations where an agent arriving at its target may block traversal for other agents. These situations previously lead to unacceptable runtimes of the CBS and exponential growth in the number of nodes in the CT.

3.3 Contemporary research on imperfect plan execution

In the final section of this chapter, we will review the recent publications that concentrate on MAPF problems where agents can suffer from imperfect plan execution. However, we must note that the problem definitions in these publications are often less complex than the Flatland. First of all, the majority of reviewed publications assume that agents have identical speeds. Secondly, they only assume agent malfunctions that last for one time-step, and lastly, they usually work on smaller grids that are open; thus, they allow for easier re-routing of agents in conflict.

We begin by reviewing Atzmon *et al.* [2018], who introduced an algorithm, which guarantees that even if each agent gets delayed by k time-steps during path execution, they will all reach its destination without colliding with other agents. These plans are called k-robust. A k-robust plan contains no k-delay conflicts, which are defined as having two agents at the same location within the time interval of $< 0, k >$ time-steps. For example, for agents with speed equal to 1 and $k = 2$, this means that they will reserve a location vertex v for 3 time-steps instead for only 1. Should any other agent plan to enter the vertex v over that time-step period, a k-delay conflict would occur; thus, the agent plans a different path.

Moreover, Atzmon *et al.* [2018] described a way to convert A^* based approaches and declarative approaches to their related k-robust versions. The conversion to k-robust plan is simple, as agents reserve vertices for longer periods depending on the chosen k. However, the core focus of the publication was the introduction of a k-robust version of the CBS (kR-CBS).

The kR-CBS prevents an arbitrarily selected agent, who takes part in the discovered conflict from entering the vertex in question over k steps. The improved kR-CBS (I-kR-CBS) version also introduced in Atzmon *et al.* [2018] imposes a k time-step long vertex entrance restriction on all the agents in the conflict individually, thus creating several children nodes in the conflict tree. These constraints can be imposed as symmetric or asymmetric as discussed in Atzmon *et al.* [2018], but the symmetric constraints were shown to provide superior performance. Atzmon *et al.* [2018] has demonstrated that the increase in the total cost of the k-optimal k-robust plans is small compared to optimal paths that assume no conflict. Finally, the symmetrically constrained I-kR-CBS outperforms kR-CBS but is still up to 20 to 50 times slower compared to the standard CBS for instances where $k = 2$, thus demonstrating the substantial increase in cost required for obtaining the robustness.

Another publication which is relevant to our problem is the Ma *et al.* [2018], which introduced algorithms that aim to maximise the number of agents completing their journey within a set time-step deadline while preventing the occurrence of agents collisions. Ma *et al.* [2018] introduced two categories of approaches for solving the problem. One category using the reduction of the problem to a flow problem and the subsequent solution using linear programming formulation of the resulting reduced abstracted multi-commodity flow network, and the other building upon search tree-based approaches. The search-based approaches included a modified version of the Conflict Based Search (CBS) and a Death-Based Search (DBS) as well as their meta-agent versions (MA-CBS, MA-DBS). Ma *et al.* [2018] found that while the reduction-based approach generally performed better on small instances with up to 50 agents, it struggled to efficiently solve large instances with a higher number of agents. For these instances, the MA-DBS and CBS were superior, with the MA-DBS offering the seemingly best performance compromise in overall.

Moreover, Hönig *et al.* [2017] studied the MAPF problem in situations where robots had kinematic constraints (i.e. velocity) and suffered from imperfect plan execution due to imprecisions in the real-world movement of robots. The publication introduced MAPF-POST approach that guarantees a safety distance between agents taking into account imperfect plan execution due to imperfections. For this cause, Hönig *et al.* [2017] introduced a temporal plan graph (TPG), which allows to avoid time-intensive re-planning when imperfect execution occurs. Temporal plan graph is created based on agents' non-conflicting path plans and its vertices represent locations that each agent is entering in its plan. Dependencies in the order of traversal of locations are identified in the TPG and safety markers are inserted into agents' plans. Agents evaluate these safety markers before entering a location to ensure that other agents, who were supposed to traverse the location before them, had done so. Hönig *et al.* [2017] then showed how to transfer the TPG into a simple temporal network, thus allowing for the usage of linear programming to solve the problem.

In another relevant publication, Ma *et al.* [2017] defined the MAPF with delay probabilities (MAPF-DP) where agents could suffer from action execution failure during every time-step. To solve the problem efficiently while preventing collisions during the execution of plans, Ma *et al.* [2017] followed up on the research done in Hönig *et al.* [2017] and utilised the TPG to control agents' plans execution. Agents' paths are planned using a modified version of the CBS Sharon *et al.* [2015a] that minimises the expected make-span of the solution while factoring in an estimation of failures in plan executions. The TPG is then constructed based on agents paths and the execution of plans is started. Every time an agent arrives at a new location, it sends messages to relevant agents that are supposed to enter the previously vacated location in the future time-steps. Recipients of this message are determined by dependencies captured in the TPG. Agents only enter the desired location if they received enough synchronisation messages from other agents; otherwise, they wait. Furthermore, to minimise the number of sent messages, Ma *et al.* [2017] performs the transitive reduction of the TPG to keep only the necessary dependencies between agents' paths.

Lastly, Atzmon *et al.* [2020] followed up on his previous research (Atzmon *et al.* [2018]) on MAPF, where agents suffer from imperfect plan execution. However, this time their research focused on finding a so-called p-robust solution. Compared with the k-robust solution Atzmon *et al.* [2018], which guarantees that agents reach their destinations even if each of them incurs up to k malfunctions, the p-robust approach guarantees that the MAPF instance is completely solved with a probability of p. Atzmon *et al.* [2020] introduced an optimal p-robust approach based on the CBS (pR-CBS) as well as a fast suboptimal algorithm based on a greedy version of the pR-CBS (pR-GCBS). Furthermore, Atzmon *et al.* [2020] showed that the kR-CBS (Atzmon *et al.* [2018]) runs faster than the pR-CBS; however, the kR-CBS neither guarantees a non-conflicting solution even with existing information about the probability of train breakage. Therefore, in general, the pR-CBS is more usable than the kR-CBS for which there does not exist any way to determine the appropriate k.

Chapter 4

Methodology

In this chapter, we will present a detailed description of approaches that we will combine into one to solve instances of the Flatland Challenge. We will first present a description of the A* algorithm as it serves as a foundation of our approach. The remainder of the chapter will then be split into two parts, where the first part will describe the approach used to plan paths for agents before an individual instance is run and the second part will detail plan execution.

This distinction is intentional as the planning approach focuses on calculating a non-conflicting and valid path for each agent while accounting for its speed and the movement of other agents. However, during the initial planning phase, there are only limited possibilities to control for unexpected train breakdown that occurs during the plan execution. Therefore, the plan execution approach described in the second section introduces a method that can be deployed on top of any planning approach in order to control for train breakdowns. A combination of the planning approach with the suitable robust plan execution method can ensure non-conflicting execution of agents' plans even if unexpected breakages occur during the plan execution.

4.1 A* path-finding algorithm

We begin this chapter by first describing the A* search algorithm (A*). The A* is an informed, best-first search algorithm similar to the uninformed Dijsktra algorithm (Dijkstra [1959]), but extended by a heuristic function, which makes the search of the state space much more efficient. We provide the pseudo-code for the algorithm in the Algorithm 3, however we note that lines 15 and 16 (highlighted in blue) are only relevant for the prioritized planning approach introduced later in this chapter.

The A* builds a tree of possible paths originating from the start node and always extends these paths one edge at a time until a termination criterion is met. The path extension order is determined by expanding nodes that minimize the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ represents the cost of travelling to the node n from agent's start node. Moreover, $h(n)$ represents the estimated cost of the cheapest path to the target node based on the heuristic function. The A* algorithm terminates when the target node is expanded or when there are no more nodes to expand.

The A* is complete when used on finite graphs with non-negative edge weights (Russell & Norvig [2009]) and optimal if the heuristic function is admissible. An admissible heuristic function is a distance function h that never overestimates the cost of getting to the target node from the current node. However, if the heuristic function is only admissible, it does not guarantee that $g(n)$ obtained upon first expansion of the node is optimal.

In order to guarantee that the $g(n)$ of a node is optimal upon its first expansion, we must require that the heuristic function is also consistent. A consistent heuristic function is such for which the following inequality holds:

$$h(n) \leq d(n, n') + h(n')$$

where the n' is a successor node of the current node n and $d(n, n')$ denotes the distance function determining the true cost of travelling from n to n'. Therefore, a consistent heuristic function does not overestimate the real cost of travelling from the current node n to its successor n'. Moreover, a consistent heuristic function is always admissible, but it may not hold the other way around Russell & Norvig [2009].

Algorithm 3 The A* algorithm

```
function A_STAR_SEARCH($agent\_id, start, target, h, CAT$)
    //h - search heuristic function, CAT - conflict avoidance table
    $open \leftarrow \{start\}$
    $closed \leftarrow \emptyset$
    $g[*] \leftarrow \infty$, $g[start] \leftarrow 0$
    $f[*] \leftarrow \infty$, $f[start] \leftarrow h(start)$
    $previous\_node \leftarrow \{\}$
    while $open$ not empty do
        $current\_node \leftarrow$ pop($open$) //Pop node with the smallest f-value
        if $current\_node == target$ then
            return rebuild_path($previous\_node, current\_node$)
        $closed \leftarrow closed \cup current\_node$
        for $neighbour \in current\_node.neighbours$ do
            if $neighbour \in closed$ then
                continue
             //Function defined in the Prioritized planning section
            if check_occupation($CAT, neighbour, time, duration$) then
                continue
            $d \leftarrow g[current\_node]+$ distance($current\_node, neighbour$)
            if $d \leq g[neighbour]$ then
                $previous\_node[neighbour] \leftarrow current\_node$
                $g[neighbour] \leftarrow d$
                $f[neighbour] \leftarrow d + h(neighbour)$
                if $neighbour \notin open$ then
                    $open.append(neighbour)$
    return Error - Path not found
end function

function REBUILD_PATH(previous_node, current)
    $path \leftarrow \{current\}$
    while current in parents_map.keys do
        $current \leftarrow previous\_node[current]$
        $path.append(current)$
    return path
end function
```

4.1.1 Heuristic functions

We now introduce Manhattan distance and Distance map heuristic functions both of which we will be using together with the A* to solve instances of Flatland Challenge. These heuristic functions are admissible and complete, hence the results provided by the individual agent A* searches will be optimal.

Manhattan distance

The Manhattan distance is an admissible heuristic suitable for 4-way grid-world environments. For two nodes i and j it is defined as follows:

$$h_{i,j} = \sum_{k=1}^{2} \mid x_{i,k} - x_{j,k} \mid$$

We note that the Manhattan distance is only consistent for grids that allow horizontal and vertical movement. However, this condition is satisfied for the Flatland grid; hence the heuristic is consistent in our case.

Distance map heuristic

The Distance map heuristic is based on a distance map of each agent. We define a distance map as a structure containing the shortest distances to the agent's target vertex from every other graph vertices reachable by the agent. For two nodes, i and j, it is defined as follows:

$$h_{i,j} = min(\text{argmin}_{\pi_{i,j}^{k}}(c(\pi_{i,j}^{0}), c(\pi_{i,j}^{1}), \dots), \infty)$$

where the $\pi_{i,j}^{k}$ denotes a $k-th$ possible path from i to j and c is a function that determines the cost of the path π.

The Distance map heuristic function is admissible as it clearly never overestimates the distance to the target node and it is trivially consistent for it never overestimates the true cost of a move to a successor node.

4.2 Prioritized planning

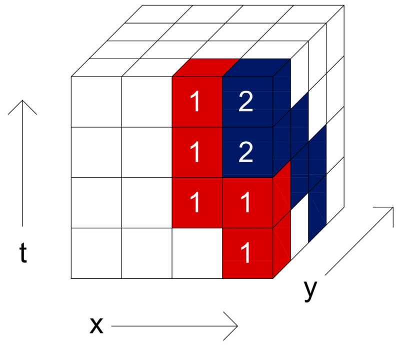
In this section, we will describe a prioritized path planning algorithm that we will deploy to plan initial paths for agents in the Flatland. However, as agents' breakdowns cannot be accurately predicted, this approach will focus on finding feasible solutions to the MAPF planning problem for agents with different speeds and ignore the possibility of agents' breakdowns.

4.2.1 Introduction of the general concept

Prioritized planning is a decoupled MAPF approach corresponding to the Cooperative A* first formalized by Silver [2005]. We plan paths for agents separately in the order determined by their priority and use the conflict avoidance table (Standley [2010a]) to prevent conflicts in planned paths. The individual agent path planning is carried out using the algorithm described in Algorithm 4.

The conflict avoidance table is a three-dimensional space-time structure (Figure fig. 4.1), which forms the core of the prioritized planning approach. Throughout the planning phase, each agent checks whether a movement to a successor tile would cause conflict with other agents and if yes, then the move is not added to the open list in the A* search. After a path for an agent had been successfully planned, all tile positions are marked as occupied in the conflict avoidance table to prevent conflict with agents planned afterwards.

Figure 4.1: Conflict avoidance table with path reservations for two agents

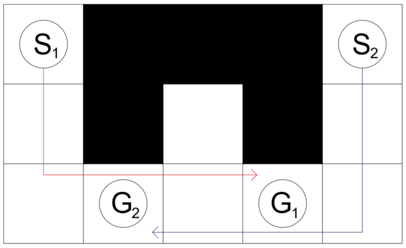


The prioritized planning algorithm is simple and fast but is not complete nor optimal. In the standard MAPF settings, where agents never leave the grid, the algorithm is incomplete as a higher priority agent can prevent other agents from entering their target positions by ending its movement in a bottleneck tile. The Figure 4.2 demonstrates a situation, where under a standard MAPF setting, the prioritized planning algorithm is incomplete regardless of agent priority assignments as the first planned agent will prevent the other from reaching its target. However, the Flatland grid starts empty, which allows placement of agents in any order as well as time-step. Furthermore, agents immediately disappear from the grid after reaching their destination; therefore, the prioritized planning algorithm is complete in this setting without further limitations imposed.

However, given the time-step constraint for each instance, the algorithm may not be able to find routes, for all priority assignments, in a way that would guarantee that all agents reach their targets on time; hence it is not complete. This highlights the fact that priority planning is sensitive to the agent priority order. Moreover, it becomes even more sensitive to it when the railways are sparse, and agents have different speeds of travel as sparse railway decreases the possibility of bypassing other agents and slow agents may hold up faster agents in bottleneck segments. For example, if we plan all slow agents first, the faster agents may not be able to finish their routes under the time-step limit as they may be held by slower agents on straight segments (e.g. Figure 4.3).

Finally, prioritised planning algorithm is a type of a decoupled MAPF approach (plans agents individually); therefore it is sub-optimal as it cannot enforce cooperation between agents that can be obtained via coupled approaches, which plan all agents together. This cooperation between agents can lead to superior plans, which are impossible to obtain via a decoupled approach such as the prioritised planning

Figure 4.2: Discussed prioritized planning issue



Algorithm 4 Prioritized planning

```
function PRIORITIZED_PLANNING($agents, h$)
    // $h$ represents a heuristic function
    $agent\_paths \leftarrow \{\}$
    $CAT \leftarrow [grid\_height][grid\_width][maximum\_time\_step]$
    $CAT[*, *, *] \leftarrow 0$
    for $agent \in agents$ do
        $agent\_path \leftarrow$ a_star_search($agent.id, agent.start, agent.target, h$)
        //Log occupation into the conflict avoidance table
        $CAT =$ log_occupation($CAT, agent.id, agent\_path$)
        $agent\_paths[agent.id] \leftarrow agent\_path$
    return $agent\_paths$
end function

function CHECK_OCCUPATION($CAT, tile, time - step, check\_duration$)
    for step ∈ range(time-step, time-step + check_duration) do
        if CAT[tile][step] != 0 then
            return True
    return False
end function

function LOG_OCCUPATION($CAT, agent.id, agent\_path$)
    for $step \in agent\_path$ do
        for $time\_step \in$ range($step.t_{start}, step.t_{end}$) do
            $CAT[step.tile.x][step.tile.y][time\_step] = agent.id$
    return $CAT$
end function
```

4.2.2 Time-expanded graph representation

The prioritised planning uses the conflict avoidance table to restrict agents' entrance to vertices v_i at the time-step t, when they are occupied by other agents. However, it is not possible to capture the element of vertex reservation at a given time t in the standard graph representation of the Flatland grid; therefore, we will now introduce the time-expanded graph.

The time expanded graph is an idea, which first appeared for SAT and domain independent approaches for MAPF, where the state representation was expanded over all possible time-steps Svancara & Surynek [2017].For our Flatland graph representation $G = (V, E, \epsilon)$ we define the time-expanded Flatland graph representation as $G_t = (V_t, E_t, \epsilon_t)$, where $t \in [0, T_{max}]$. Vertices V_t correspond to the $t - th$ layer of the graph, thus represent the grid in $t - th$ time-step. Furthermore, E_t contains the traversal edge $\epsilon(e) = (v_t, u_{t+\rho})$, where ρ represents the number of time-steps an agent needs to traverse a tile, if and only if E contains the edge $\epsilon(e) = (v, u)$. Similarly, the E_t contains the waiting edge $\epsilon(e) = (v_t, v_{t+1})$ if and only if E contains the edge $\epsilon(e) = (v, v)$.

For simplicity we assume that given the dependency on ρ, we need to construct as many time-expanded graphs for the prioritised planning as we have different ρ values for our agents. Therefore, different agents will be using different time-expanded graphs for the path planning; however, all resulting paths will be noted into the shared conflict avoidance table afterwards. Hence, we will run the individual agents A* search on the time-expanded graph and when an agents expands a node, it will look-up occupancy in the conflict avoidance table based on the t of the expanded vertex.

4.2.3 Agent priority heuristics

As explained above the prioritized planning approach is sensitive to the priority assignments to agents and unsuitable priority assignments can deteriorate the performance of the approach and cause partial completion of paths under time-step constraint. We will now introduce several agent priority ordering heuristics in order to try to boost the performance of the prioritised planning algorithm.

Before we outline our heuristics, we remind the reader that we defined the number of time-steps an agent requires to traverse a tile, based on its speed, in the Chapter 2 in the Equation 2.1.

While developing our agent priority heuristics we have taken into account several key learnings resulting from principles of the Flatland Challenge. The key learnings used for the definition of the ordering heuristic are the following:

Learning 1: The railway in Flatland Challenge is sparse and there exists

> only a limited number of target destinations that accommodate multiple agents. This results in agents often following one another on long straight railway segments. Thus, a train queue lead by one slower agent could hold-up several fast paced agents on these segments, thus severely affecting the overall performance. Therefore, higher speed agents should be planned with a higher priority. On the other hand, if we schedule slower agents too late, they may not have enough time to reach their target before the time-step limit for the Flatland instance runs out.
>
> **Learning 2:** In the Flatland Challenge, all agents have a unified probability of breaking down during each time-step. Due to the speed difference affecting the number of time-steps required to traverse a tile, slower agents are more prone to breakages on paths of the same length. This observation further supports the conclusion from the previous learning that higher speed agents should be planned with a higher priority. On the other hand, if slow agents are scheduled too late, they may not be able to reach their target. This is only worsened by the fact that they may breakdown hold up themselves and potentially other slower agents.
>
> **Learning 3:** Agents with the same speed, but longer paths are more prone to breaking down as they require more time-steps to complete their journey. Therefore, agents who start closer to their destinations should be scheduled first. On the other hand, we again risk scheduling agents with long paths too late, which may result in deterioration of completion.

From our learnings we observe that there does not seem to be a one apparent solution to the agent ordering problem in the Flatland. This claim is further supported by the fact that both agents' start/target positions and speeds are randomly distributed at the start of each distance, hence there does not exist any fixed logic of agents' speed and distance to goal assignments. Therefore, we will now introduce 4 agent ordering heuristics and we will analyse their performance in the Chapter 5 when we conduct repeated experiments on several Flatland instance.

We split our new ordering heuristics into two categories. The first category prioritizes agents who have a generally faster path completion and we call it Upwards ordered heuristics as the completion times of scheduled agents gradually increase. The other heuristic category prioritises agents with a longer path completion time and we call it Downwards ordered heuristics. Our ordering heuristics are defined as follows:

Upwards ordered heuristics

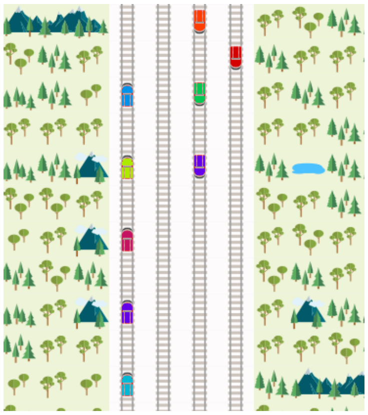
- Fast-First ordering
    1. Order agents by the number of time-steps needed to travel a tile in an increasing order
    2. Break ties by prioritising agents with shorter distance to their target position
    3. Agents with the same speed and distance to their target are ordered by their initial Flatland priority in an increasing order
- Close-First ordering:
    1. Break ties by prioritising agents with shorter time-step distance to their target
    2. Agents with the same speed and distance to their target are ordered by their initial Flatland priority in an increasing order

Downwards ordered heuristics

- Slow-First ordering:
    1. Order agents by the number of time-steps needed to travel a tile in a decreasing order
    2. Break ties by prioritising agents with longer distance to their target position
    3. Agents with the same speed and distance to their target are ordered by their initial Flatland priority in an increasing
- Remote-First ordering:
    1. Break ties by prioritising agents with longer time-step distance to their target
    2. Agents with the same speed and distance to their target are ordered by their initial Flatland priority in an increasing

Where the time-step distance represents the lowest number of time-steps the agent needs to reach its target. This value is obtained by multiplying agent's tile distance from the target by the number of time-steps the agent requires to traverse a tile. We have done this to directly factor in the agent's speed into the ordering. We note that in the case of the Fast-First and Slow-First orderings, the distance can be represented as the number of tiles that an agent has to traverse to reach its target as we are always breaking tiles between agents with the same speed.

Figure 4.3: Convoy of agents on a railway straight segment



4.3 Dynamic traversal precedence maintenance

In this section, we will introduce an approach that can be used to prevent conflicts caused by the occurrence of train breakdowns during the execution of plans. We will restrain from implementing approaches that rely on repeated re-planning of agents' path after breakdown occurs. Although these approaches can, when possible, improve routes after malfunction happens, they are computationally very costly, especially for large instances where breakdowns are frequent.

Therefore, our aim is to dynamically prevent conflicts when an agent breakdown occurs, while keeping the computational time independent of the frequency of breakdown occurrence.

4.3.1 Introduction of the general concept

We will now introduce a dynamic path execution approach that is deployed during the path execution phase and can be used with any path planning algorithm.

The approach is built around a temporal plan graph (Hönig *et al.* [2017]) that captures the inter-dependencies between agents' paths. Specifically, we will use it to establish and enforce the order of the agent's traversal through tiles. Our approach was inspired by Hönig *et al.* [2017], Hönig *et al.* [2019] and Ma *et al.* [2017], but we implemented improvements allowing trains to skip redundant wait moves.

The main benefit of this approach is that the agent paths can be planned without assuming agent any breakdowns. Afterwards, during the path executions, we enforce the tile traversal order that prevents agents conflicts without a need for further re-planning.

4.3.2 Temporal plan graph definition

The temporal plan graph (TPG, Hönig *et al.* [2017]) is a directed acyclic graph $G_{TPG} = (V_{TPG}, E_{TPG})$, where vertices $v_i \in V_{TPG}$ correspond to agent's entry into a Flatland grid tile and edges $e_i \in E_{TPG}$ represent the temporal precedence between events represented by vertices v and v'.

The TPG imposes two types of precedence constraints:

1. Agent a_i must enter grid positions in the order imposed by its path plan
2. Agents must obey the imposed order of traversal for grid positions that are traversed by multiple agents

The time-step factor is not directly captured in the TPG; however, given the precedence constraints, the TPG specifies a partial order among events Ma *et al.* [2017]. This allows us to execute agents' paths without conflicts even when breakdowns occur as Hönig *et al.* [2017] proved that agents do not collide if they execute a movement schedule that is consistent with the TPG's precedence constraints.

4.3.3 Temporal plan graph construction

After paths for agents have been found, we will encode them into the TPG. We first create a directed graph out of agents' individual paths and then we find the precedence constraints and represent them as edges between vertices in individual agents paths. The construction of the TPG is in detail described in the Algorithm 6. During the construction of the TPG, we will leave out all the wait moves planned for all agents as they become a redundant complication because agents will always have to wait until all precedences for their next movement are satisfied. The absence of wait moves in the TPG allows us to, in some cases, compensate for time lost due to the agent's breakdown. For example, an agent may have 5 consecutive wait movements lasting 5 time-steps scheduled at some point, but if it endures a breakage lasting for 3 time-steps before it is supposed to execute the 5 wait moves, it is sufficient for it to execute only 2 wait moves to preserve all precedences, ceteris paribus[1]. Moreover, the precedence edges between vertices corresponding to one agent carry the information about the action that the agent needs to execute to get its next position, thus encoding the plans for all agents.

The Algorithm 6, with un-commented line above the line 20 and removed current line 20, describes the baseline TPG construction algorithm described in Hönig *et al.* [2017] and **?**. However, this baseline version is unnecessarily complex, which is especially problematic as our implementation is in Python. Therefore, we will introduce a significant improvement to the construction of the TPG, which leverages the conflict avoidance table (CAT) that we created during the path planning phase.

Specifically, the baseline Algorithm 6 assumes iteration over all other agents (comment above the line 20). However, this is inefficient as many of these agents do not have any precedence with the tile as they do not even traverse it in their plans. To solved this, we use the conflict avoidance table that helps us to determine exactly which agents traverse the tile in question after the agent in question (Algorithm 5. Therefore, we then iterate over only a subset of the agent space as seen on the Line 20 in the Algorithm 6.

Moreover, we note that the TPG construction improvement is applicable to output of any non-conflicting path finding algorithm as the conflict avoidance table can be quickly filled in after the path finding with a different algorithm (such as CBS) done. Therefore, we obtain similar benefit as if we used the prioritised planning. This improvement leads to a substantial reduction in the run-time of the construction of the TPG, which we will demonstrate in the Chapter 5.

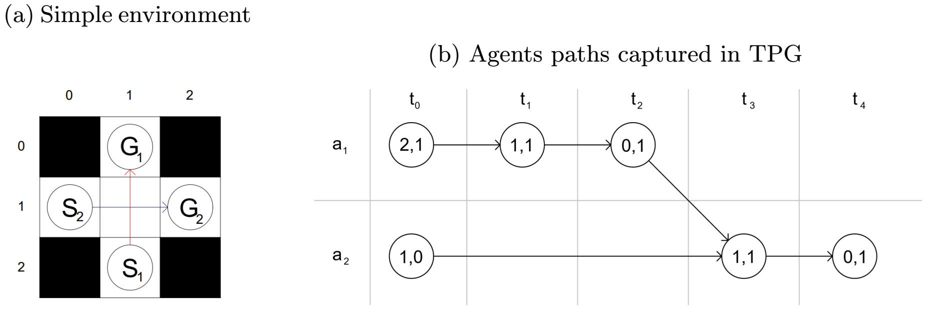
[1] Other things equal

4.3.4 Temporal plan graph usage

We will use the TPG during the execution to monitor whether all required agents have already traversed the vertex, thus allowing for the next in line agent to start traversing it. The precedence can be easily enforced via a central planner or a message communication system between agents; we will use the latter approach. Throughout the execution, after an agent enters a new tile, it sends a message to all agents that are supposed to later traverse the node the agent just left. The precedence conditions for a tile entry of an agent are satisfied when the traversal precedence counter for its next TPG vertex equals to $k-1$, where k is the number of in-edges, and we deduct the 1 as agents are not marking traversals of vertices along their own path.

To be able to prevent all collisions, we need to make sure that agents maintain safety distance. The safety distance is enforced by the fact that agent increments the traversal precedence counter for a TPG vertex if and only if it had already successfully traversed the corresponding tile. Hence, we only permit the agent to start traversing a node if all the previous agent in the precedence order had successfully traversed it. The downside of the safety distance is that it enforces spacing between agents equal to one additional time-step. This additional time-step may otherwise not be necessary given the Flatland agent movement execution, which performs agents' actions one by one according to their Flatland ID, which is defined arbitrarily. Therefore, in some situations, the following conflict (as defined in Chapter 2 does not occur. However, the TPG plan execution ignores this opportunity to save one time-step to maintain the safety distance preventing collisions.

Figure 4.4: Temporal plan graph construction logic


(a) Simple environment

(b) Agents paths captured in TPG

Lastly, in the Figure 4.4 we provide an example of TPG representation of agents' paths. Figure 4.4a shows a simple MAPF situation, where S_i and G_i represent the starting and target positions for the agent a_i. The agent a_1 has a higher priority, thus will be planned as the first one. The TPG representation of agents paths is shown in Figure 4.4b. Rows in Figure 4.4b correspond to individual agents paths, where the wait moves of the agent a_2 at t_2 and t_3 are already removed. The edge connecting the node $(0, 1)$ of the agent a_1 to the node $(1, 1)$ of the agent a_2 represents a precedence relationship as agent a_2 can only enter the node $(1, 1)$ after the agent a_1 leaves it. Therefore upon leaving the node $(1, 1)$ and arriving into the node $(0, 1)$ the agent a_1 notifies the agent a_2 via the precedence edge that it can start traversing to the node $(1, 1)$.

Algorithm 5 Temporal plan graph improvement

```
function SELECT_AGENTS($agent\_id, tpg\_vertex, CAT$)
    $agents \leftarrow \emptyset$
    for $t \in range(tpg\_vertex.t + 1, T\_max)$ do
        if $CAT[tpg\_vertex.x, tpg\_vertex.y, t] \notin agents$ then
            $agents \leftarrow agents \cup CAT[tpg\_vertex.x, tpg\_vertex.y, t]$
    return $agents \setminus agent\_id$
end function
```

4.4 p-TPG

We will now introduce our approach for solving the Flatland instances, which is composed from the above defined methodology. From now on, we will call the approach "p-TPG".

The p-TPG is a two-stage algorithm that first plans non-conflicting paths for all agents using the prioritised planning, which runs a low-level A* search with a conflict avoidance table. In the second stage, it creates the temporal plan graph based on paths planned in the first stage. The temporal plan graph then controls the plan execution through the dynamic traversal precedence maintenance and allows us to effectively handle agent breakdowns.

Algorithm 6 Temporal plan graph construction

```
function TEMPORAL PLAN GRAPH(agents, map_graph)
    $G_{TPG} \leftarrow (E_{TPG}, V_{TPG})$
    for agent ∈ agents do
        $i \leftarrow agent.id$
        //agent's starting ver
        $v_0^i \leftarrow agent.path[0].ver$
        $V_{TPG} \cup v_0^i$
        $v_{last} \leftarrow v_0^i$
        for j in range(1, length(agent.plan)) do
        //ignore wait actions
            if agent.path[j].ver != agent.path[j-1].ver then
                $t \leftarrow agent.path[j].time$
                $v_t^i \leftarrow agent.path[j].ver$
                $V_{TPG} \cup v_t^i$
                $action \leftarrow extract\_action(map\_graph, v_{last}, v_t^i)$
               //add edge with an action note
                $E_{TPG} \cup (v_{last}, v_t^i, action)$
    for agent ∈ agents do
        $i \leftarrow agent.id$
        //The last node in plans is a sink node
        for k ∈ range(1, length($agent.plan$) − 1) do
            $t_i \leftarrow agent.plan[k].time\_step$
            if $v_t^i \in G_{TPG}$ then
              // for  agent_j ∈ agents \ agent   do
                for $agent\_j$ ∈ select_agents($i, v_t^i, CAT$) do
                    $j \leftarrow agent\_j.id$
                    for $t_j \in range(\mathrm{t}_i + 1$, length($agent.plan$)) do
                        if $v_{t_j}^j \in G_{TPG}$ then
                            if $agent.plan[t_i].ver == agent\_j.plan[t_j].ver$ then
                             //precedence edge from agent's next node
                                $E_{TPG} \cup (v_{t+1}^i, v_{t_j}^j)$
                                $break$
    return $G_{TPG}$
end function
```

Chapter 5

Experiments and Results

In this chapter, we will analyse the outcomes of testing of our Flatland solving MAPF approach. We will begin the chapter by describing the testing methodology, including the introduction of our Flatland test instances alongside the agent speed and breakdowns generators. We will then begin to review the results of the test we have conducted. First of all, we will examine the effects of different state-space search heuristics on the prioritised planning approach introduced in the Chapter 4. Then we will describe the effects that different agent priority heuristics have on the performance of the prioritised planning. Lastly, we will analyse the performance of our p-TPG approach on instances with breakdowns.

We note that we ran all calculation using a workstation equipped with the Intel Xeon e3-1240 v3 clocked at 3.40 GHz and 8GB of DDR3 RAM clocked at 1600MHz. None of our calculations were in any part done using a GPU.

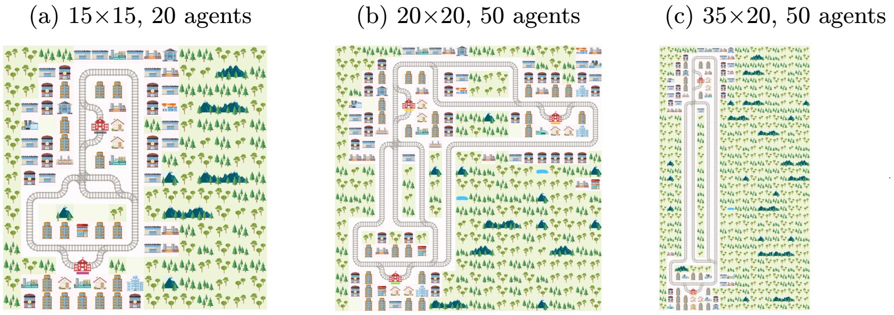
5.1 Flatland test instances

Through iterative development, we have developed instances that are representative of environments that we had seen in the actual Flatland Challenge[1] and also gradually increase in complexity to allow for better analysis of the performance of our approach.

[1] Leaderboards contain videos from the competition simulation run-time showing Flatland environments, accessible at https://www.aicrowd.com/challenges/flatland-challenge/leaderboards

In total, we have prepared 12 test instances, which are split into 4 categories: Easy shown in the Figure 5.1, Medium shown in the Figure 5.2, Hard shown in the Figure 5.3 and Extreme shown in the Figure 5.4. The categorisation is determined by the difficulty of solving an instance, which is driven not only by the size of the environment and the number of agents but also by the number of cities and routes between them. Instances in each category should be progressively increasing in difficulty. However, there are many variables interacting and driving the complexity; thus, deviations from the general trend of increasing complexity are possible.

Figure 5.1: Easy test instances



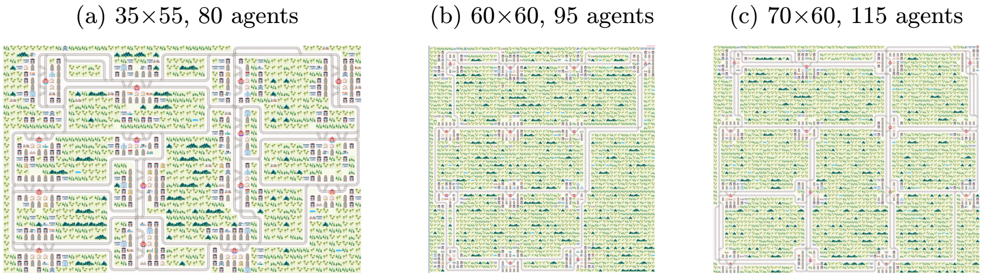
(a) 15×15, 20 agents

(b) 20×20, 50 agents

(c) 35×20, 50 agents

We note that the AI Crowd provided 20 test cases SBB, AI Crowd [2020], which we took into account while creating our test instances. However, these test cases were mostly of medium or extreme difficulty and had a very infrequent occurrence of breakdowns; hence they were not very suitable for a detailed analysis of the performance of our approach across various situations.

Figure 5.2: Medium test instances



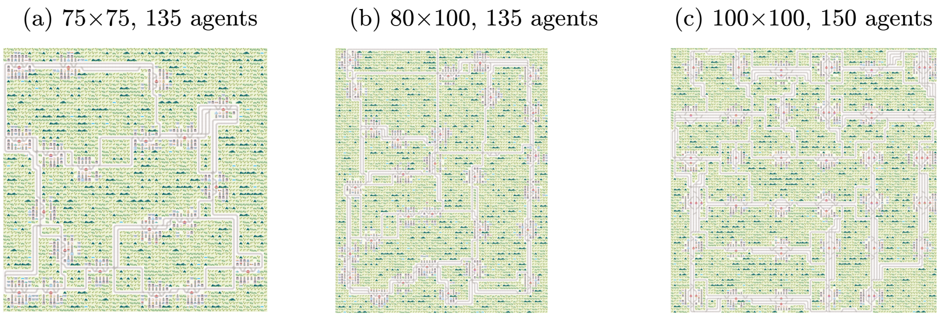
(a) 35×55, 80 agents

(b) 60×60, 95 agents

(c) 70×60, 115 agents

When examining the Figures depicting railway maps of our test instance, we would like to bring to attention that all our test instances have very sparsely distributed railways with long straight segments (without bypasses) and a very large number of agents, which is possible only because the placement of agents on the grid is part of the planning process and agents disappear after reaching their target position. In the end, these two properties lead us to prefer the sub-optimal prioritised planning approach over other optimal approaches, mostly due to the much faster speed of finding a feasible solution.

Figure 5.3: Hard test instances

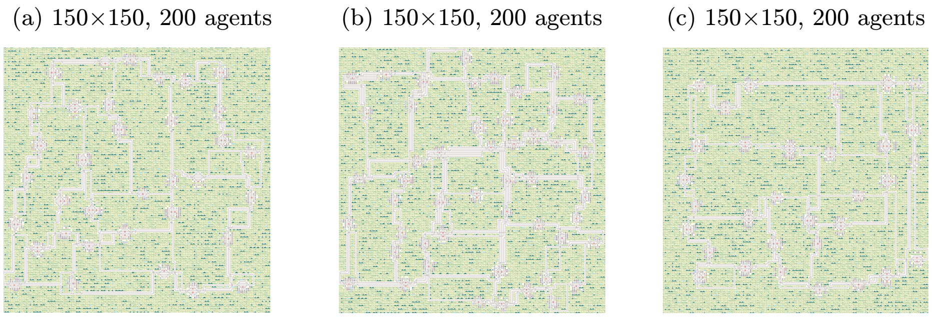

(a) 75×75, 135 agents

(b) 80×100, 135 agents

(c) 100×100, 150 agents

The fact that agents disappear after reaching their targets makes it easier to use prioritised planning as one agent can never block others by occupying its target position for the rest of the instance. However, the need to place agents on the grid makes it especially difficult to solve the Flatland instances with coupled approaches (OD+ID by Standley [2010b]) or tree-based approaches (CBS by Sharon *et al.* [2015b]) in their elemental form. This is because the number of potential conflicts among agents is very high because of their shared placement positions. Moreover, the CBS especially struggles when dealing with conflicts of agents who are travelling in opposite directions across long straight rail segments.

Figure 5.4: Extreme test instances


(a) 150×150, 200 agents

(b) 150×150, 200 agents

(c) 150×150, 200 agents

5.1.1 Agent speeds

Having created railway maps, we next need to determine the range of speeds that different agents can have in these environments. In line with the Flatland Challenge, we are using 4 different speeds for trains representing the following real-world categorisation: fast passenger train with speed equal to 1.0, fast cargo train with speed equal to 0.5, slow passenger train with speed equal to $\frac{1}{3}$ and slow cargo train with speed equal to 0.25. Each agent is assigned one of the above-introduced speed categories with a uniform probability of 0.25 each.

5.1.2 Agent breakdown generators

Lastly, we need to determine test configurations of agent breakdown generators. We remind the reader that we provided a detailed explanation of the breakdown mechanics in Chapter 2. The Flatland Challenge environment allows for great flexibility in breakdowns configurations as we can not only determine the probability of an agent breaking down during each time-step, but we can also set a minimal and maximal time-step duration for breakdowns. However, we highlight that all trains share the same probability of breaking down during each time-step.

Unlike for the agent speed distribution, The Flatland Challenge provided no specific information about the setup of breakdown generators except for the breakdown generator in the public test instances. Therefore, we have defined 3 breakdown scenarios for our testing, which we are as follows:

- Frequent: Frequent breakdown occurrence / short breakdown duration
- Moderate: Moderate breakdown occurrence / medium breakdown duration
- Rare: Infrequent breakdown occurrence / long breakdown duration

Moreover, we provide further details about these scenarios in the Table 5.1, where we specify the probability of train breakdown during each time-step, the range of possible breakdown durations, and the expectation indicator. We define the expectation indicator as follows:

$$\textit{Expectation} = \textit{Probability} * \textit{Average duration}$$

and it represents the average expected breakdown duration in each time-step. We use this indicator to ensure that our breakdown scenarios will lead to a similar cumulative breakdown durations when deployed on the same test instance. Therefore, we can isolate the effect that different average breakdown durations have on our approach performance.

Table 5.1: Breakdown scenarios properties

Scenario	Probability	Possible duration	Expectation
Frequent breakdowns	0.0043383	[2,5]	0.015
Moderate breakdowns	0.0009995	[10,20]	0.015
Rare breakdowns	0.0003999	[25,50]	0.015
Flatland example	*0.0000833*	*[20,50]*	*0.00291*

Lastly, the last row in the Table 5.1 illustrates how the breakdown generator was configured for the public test instances provided by the AI Crowd SBB, AI Crowd [2020]. However, as these instances were meant for fine-tuning of contestants algorithms, we did not consider the configuration to be challenging enough and increased the difficulty in our scenarios, which can be seen based on the difference in the expectation indicator.

5.2 Prioritised planning performance analysis

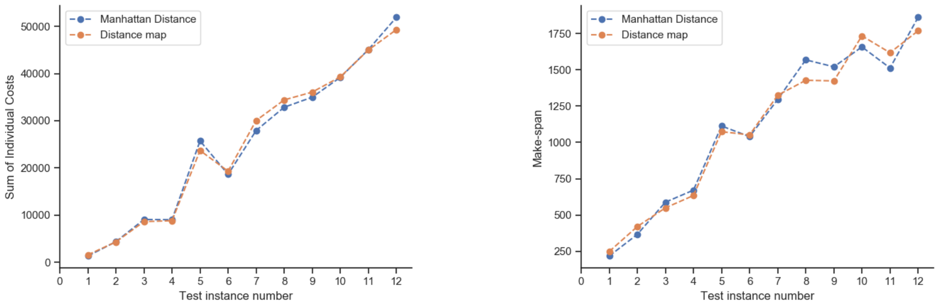
In this section, we will first examine the effects that the choice of a heuristic function can have on the efficiency of the state space search in the prioritised planning approach. Next, we will analyse the impact of agent priority heuristics introduced in Chapter 4 on the overall outcomes and efficiency of the prioritised planning. To isolate the effects of these heuristics on the planning results, we will solve our test instances without any breakdowns occurring.

5.2.1 State space search heuristics comparison

The prioritised planning approach is built around the A* algorithm, which we use to plan non-conflicting paths for individual agents. Therefore it is essential to make the approach as fast as possible by using the most efficient state search heuristics. Thus, we will now compare the efficiency of the Manhattan distance and Distance map search heuristics, which we introduced in Chapter 4.

As outlined above, we ran the prioritised planning on our test Flatland instances, but ignored any breakdowns. The results we obtained are summarised in Figures 5.5, **??** and 5.9.

Figure 5.5: The effect of state space search heuristics on paths length

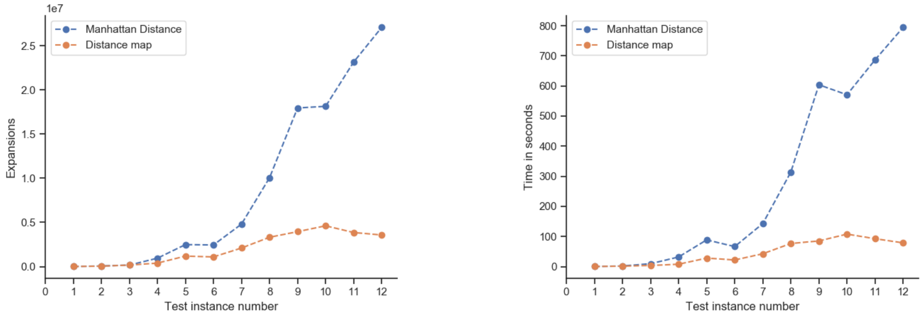


(a) Sum of Individual Cost

(b) Make-span

In the Figure 5.5 we see that while both heuristics lead to almost identical results in terms of the sum-of-individual cost (SIC, Figure 5.5a), they differ slightly in the Make-span (MS) on more complex instances (Figure 5.5b). This phenomenon may surprise some readers as they may have expected the results of the two heuristics to be completely identical. However, several optimal paths may exist for some agents in the Flatland grid, and the choice of the path for one agent may affect the path planning for succeeding agents. Thus, resulting in differences in the overall plan, which causes differences in the SIC and MS.

Figure 5.6: The effect of state space search heuristics on performance

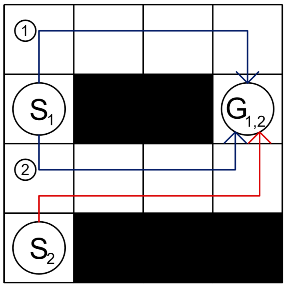


(a) Number of A* state expansions

(b) Planning run-times

To demonstrate this hierarchical effect, we provide an example supporting our reasoning in Figure 5.7. Agent 1 has a higher priority and two optimal routes leading to its target position. However, should it choose the second optimal path, leading through the bottom of the environment, it would result in a longer path for the second agent, which would have to prevent a tile conflict (Chapter 2) with the first agent on its optimal path. Therefore, the decision about the path of the first agent can increase the SIC as well as the MS of the instance solution.

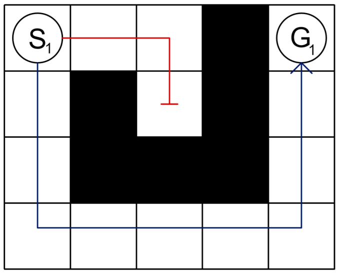
Figure 5.7: Hierarchical dependency of path planning



Moreover, Figure 5.6 depicts the cumulative number of nodes expanded during all individual searches and run-times of the overall search in seconds across our test instances for both heuristics. We see that the Manhattan distance leads to a much higher number of expansions on hard and extreme instances, which, as expected, directly affects the overall run-time of the state space search through the increase in the search complexity. Using the Manhattan distance resulted in up to 6 times more state expansions and longer search on extreme instances when compared to the results obtained using the Distance map heuristic.

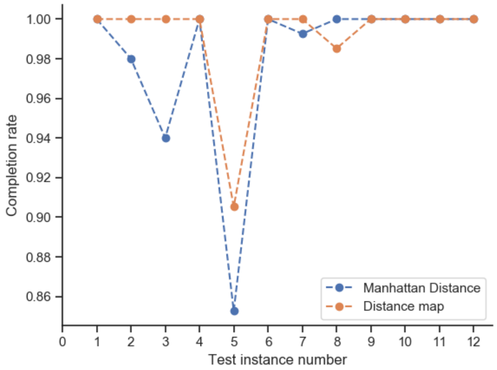
The difference in expansion count is caused by the fact that the Manhattan distance generally has less accurate information about the real structure of the Flatland grid compared to the Distance map, which maps true distances based in the grid. Figure 5.8 depicts an example, where using the Manhattan distance will lead to more expansions and longer search than using the Distance map. In the Figure 5.8, we observe that the Distance map heuristic will start expanding the blue path leading to the target position right from the start. However, the Manhattan distance will first cause additional expansions depicted by the red line before expanding the feasible way to the target position. These additional expansions are entirely misleading, but the heuristic has no information that would help to determine that sooner as according to the values of the Manhattan distance, the agent should be approaching the target position when travelling along the red path.

Figure 5.8: Manhattan distance node expansion inefficiency



Lastly, in the Figure 5.9, we summarise the effect that the two state-space search heuristic functions have on the completion rate of our test instances without breakdowns happening. As explained above, although both heuristics guarantee the optimality of individual agents' paths, the overall prioritised planning solution can differ. As there are no breakdowns, any partial completion is caused by the fact that some agents are unable to reach their positions under the time-step limit specified for the test instance. From the Figure 5.9 we see that both heuristics lead to a partial completion on the test instance 5, which seems to be more challenging to complete than the other test instances. Furthermore, using the Distance map heuristic leads to a 2% drop in completion for the test instance 8. However, the planning with Manhattan distance is only partially complete on test instances 2, 3 and 7. In general, we observe that the Manhattan distance resulted in worse results in terms of the completion rate than using the Distance map heuristic.

Figure 5.9: The effect of state space search heuristics on completion rate



In overall, while both heuristics offer comparable results in terms of SIC and MS;

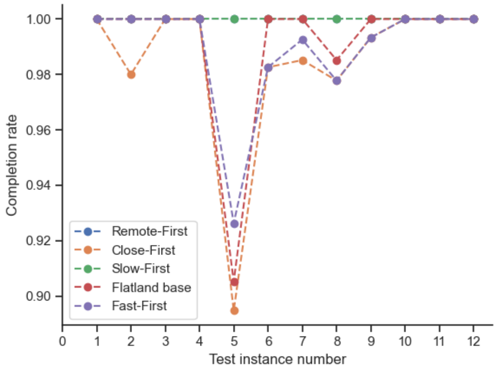
using the Manhattan distance leads to lower average instance completion rate and significantly worse run-time of the prioritised planning algorithm. Therefore, from now on, we will only use the Distance map heuristics.

Agent ordering

In the Chapter 4, we have explained that the prioritised planning algorithm is sensitive to the order in which we plan paths for individual agents. Therefore, we will now analyse the impact that the different agent priority heuristics, which we introduced in the Chapter 4, can have on the prioritised planning approach.

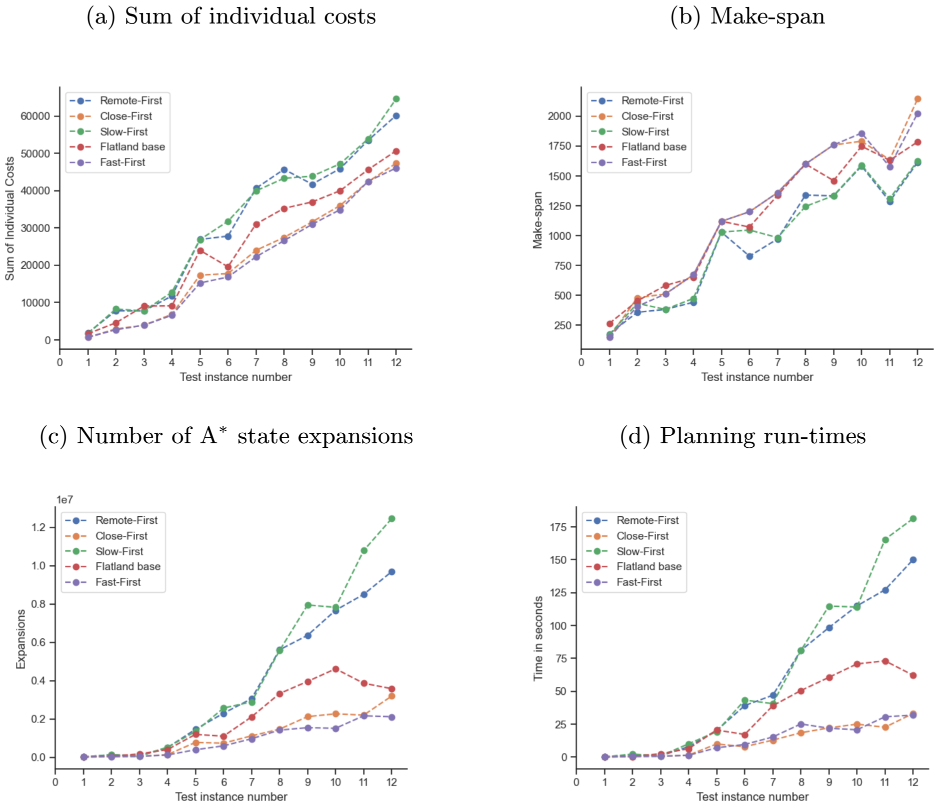
Figure 5.10 summarises the impact of different ordering approaches on the completion rate in simulations where breakdowns do not occur. Interestingly, we see that approaches that prioritise agents with shorter path execution in terms of time-steps (Fast-First, Close-First) provide a sub-par performance comparable to the initial Flatland priority assignment. On the other hand, approaches prioritising agents with prolonged path execution in terms of time-steps (Slow-First, Remote-First) maintain full completion rate.

Figure 5.10: Test instances completion rate without breakdown



Moreover, figures 5.11a and 5.11b depict the effect of various ordering heuristics on the total execution cost measured by SIC and MS as well as on the performance of the state space search. From Figure 5.11, we see that there exists an inverse relationship between the SIC and MS of individual ordering approaches. While the Fast-First and Close-First orderings dominate other orderings in terms of SIC, they provide a sub-par performance in terms of the MS compared to the other ordering heuristics. On the other hand, Slow-First and Remote-First ordering heuristics perform very well in terms of the MS, but have a much higher SIC.

Figure 5.11: Testing results without agent breakdowns



(a) Sum of individual costs

(b) Make-span

(c) Number of A* state expansions

(d) Planning run-times

The reasoning behind this inverse relationship is relatively trivial. Fast-First and Close-First orderings prefer to first schedule agents, who will complete their routes quickly. These approaches, therefore, have a lower SIC as fast agents quickly complete their routes and then do not accumulate any additional cost. However, as agents who need more time-steps to complete their routes are scheduled with lower priority, they increase the overall time-step duration of the instance; thus, increase the make-span. Furthermore, this phenomenon causes the decrease in the completion rate for these orderings as the slowest agents may not have enough time to complete their routes when scheduled close to the end of the instance determined by the time-step constraint. Moreover, it can be trivially seen that the same logic in reverse applies to Slow-First and Remote-First orderings. First of all, agents with longer paths in term of time-steps have enough time to complete their routes and can be closely followed by fast agents; therefore result in lower MS. However, the generally higher number of waiting agents in the environment leads to the accumulation of additional costs; thus increase the SIC.

Lastly, Figures 5.11c and 5.11d show that Remote-First and Slow-First heuristics have a significantly higher count of search space expansions and longer planning run-times on hard and extreme test instances when compared to all ordering approaches. On the other hand, Fast-First and Close-First ordering heuristics have a low count of expansions and consistently short run-times, which again becomes apparent on hard and extreme instances, where they over-perform all other agent orderings in terms of these metrics. Specifically, the count of expansions and run-times for the most computationally demanding agent ordering, Slow-First, are up to 7 times higher when compared to the Fast-First agent ordering that was the least computationally demanding ordering.

The difference in expansions and the search run-time occurs because agents that require more time-steps to reach their target positions are scheduled first, which then imposes more restrictions on the path planning for lower priority agents. These planning restrictions are caused by the fact that agents moving slowly or/and over longer distances block out more positions in the conflict avoidance table, thus limiting options for the planning of succeeding agents. Lower priority agents, regardless of their speed then require a deeper and longer search to find an optimal and valid path. We note that the increase in expansions is primarily caused by expanding wait moves, which agents schedule either before placement or while waiting for other previously scheduled agents to complete their traversal of agent's best succeeding position.

In overall, the completion rate is the key indicator of success in terms of Flatland and we found that the Slow-First and Remote-First agent ordering not only dominate the Flatland initial priority assignment but also offer superior results when compared to the Fast-First and Close-First orderings. Moreover, in terms of the completion rate, the Fast-First and Close-First orderings in general offered a sub-par performance even when compared to the Flatland initial agent ordering. However, the apparent downside of Slow-First and Remote-First agent orderings is the higher complexity of the planning process reflected in the higher number of expansions and longer run-times on more complex test instances.

5.3 Testing with breakdowns

In the last section of this chapter, we will evaluate the effect of the three breakdown scenarios we introduced at the beginning of this chapter on the results of our approach. We remind the reader that the three breakdown scenarios are as follow:

- Frequent: Frequent breakdown occurrence / short breakdown duration
- Moderate: Moderate breakdown occurrence / medium breakdown duration
- Rare: Infrequent breakdown occurrence / long breakdown duration

where all have almost the same expected duration of breakdown per time-step in order to make them more comparable. To make our results more robust, we ran each instance 30 times, each time with a different random seed for breakdowns and then averaged out the results.

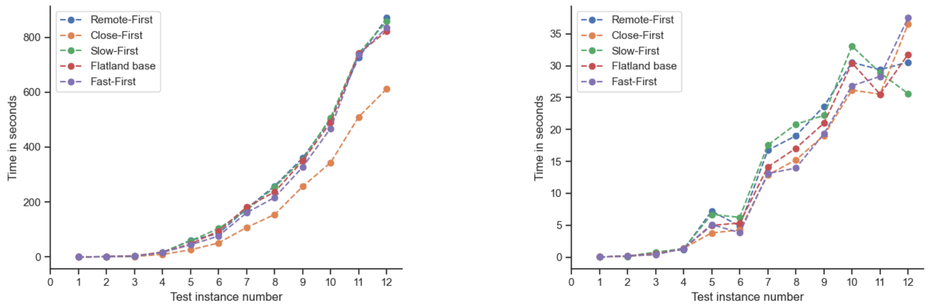
We note that we will not further analyse the count of expansions and planning run-times as they are identical as for the cases with no breakdowns, which we described in Figures Figures 5.11c and 5.11d. This is because we use the prioritised planning to plan agents' paths, ignoring the existence of breakdowns, and then we construct the TPG to control for breakdowns occurrence during paths execution.

5.3.1 Temporal plan graph construction improvement

In Chapter 4, we introduced an improvement to the baseline TPG construction algorithm (Hönig *et al.* [2017]), which benefits from using the conflict avoidance table from the prioritised planning, introduced in the Chapter 4. The Figure 5.13 illustrates the impact of the improvement on the TPG construction algorithm run-time. When comparing Figures 5.12a and 5.12b, we observe that the improvement we introduced provides decreases in the TPG construction run-time, which can account for more than 2.5 orders of magnitude for some extreme instances.

The improvement in the run-time of the TPG leads to a major reduction in the total planning run-time of our approach. The total planning run-time represents the sum of prioritised planning and the TPG construction run-times. Figure 5.13a depicts the total planning run-time when the baseline TPG construction

Figure 5.12: TPG construction algorithms run-times comparison

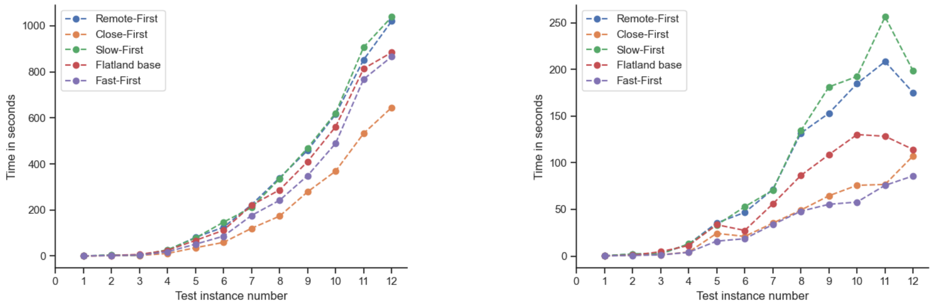


(a) Baseline TPG construction run-times

(b) Improved TPG construction run-times

algorithm was used, while Figure 5.13b shows the total planning run-times with the improved TPG construction algorithm. We observe that the fastest planning approaches, using the Fast-First and Close-First agent ordering, now only take up to 100 seconds compared to up to 800 seconds with the baseline TPG construction algorithm; thus, reducing the total planning run-time 8times. The total planning run-time with the Slow-First and Remote-first orderings is about 4 times lower too as well. However, in this case the reduction of run-time is lower as the path planning accounts for a larger part of the planning run-time for these two orderings.

Figure 5.13: Total planning run-times comparison



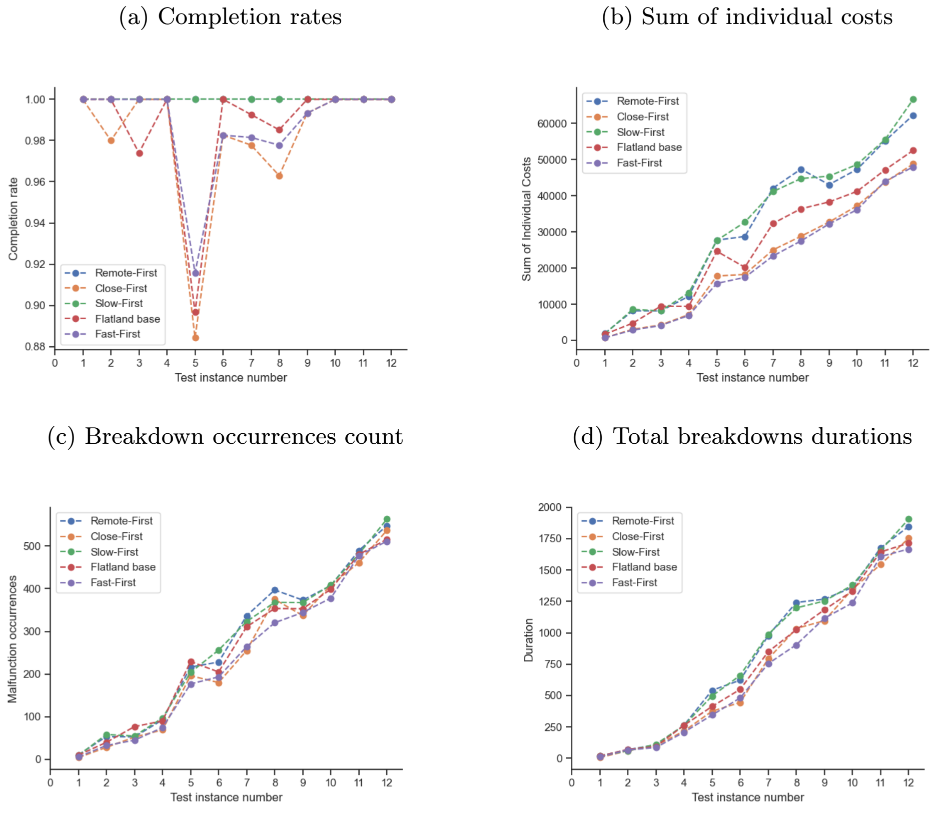
(a) Baseline total planning run-time

(b) Improved total planning run-time

5.3.2 Breakdown scenarios results

We will now review the complete results from running our approach on instances with the breakdown scenarios. The results for the scenario with Frequent breakdowns can be seen in the Figure 5.14, for the scenario with Moderate breakdowns in the Figure 5.15 and for the scenario with Rare breakdowns in the Figure 5.16. From our results, we can see that in general, the impact of breakdowns was the least significant for the scenario with Frequent breakdowns and the most severe for the scenario with Rare breakdowns.

Figure 5.14: Testing results obtained with Frequent breakdowns



(a) Completion rates

(b) Sum of individual costs

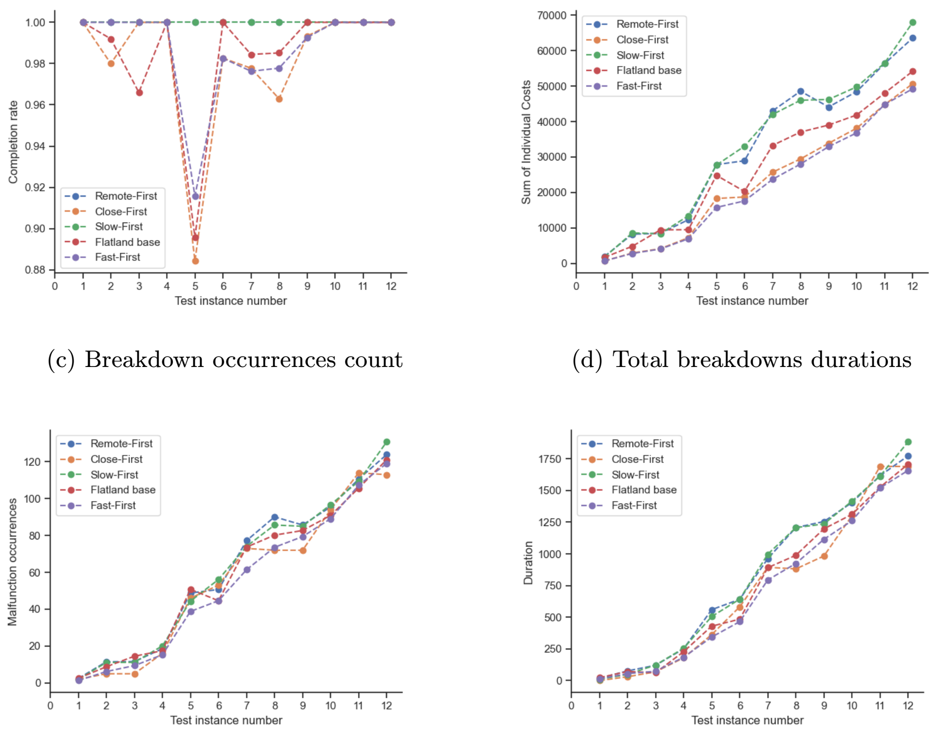
(c) Breakdown occurrences count

(d) Total breakdowns durations

The completion rate in the scenario with Frequent breakdowns was affected mostly for the Flatland initial ordering and Close-First ordering, which registered additional 2% completion rate decreases for already incompletely solved instances as well as new decreases in completion for the Easy test instance number 3 and 7. Moreover, the scenario with Moderate breakdowns continued in this trend but mostly affected the plan created using the Flatland initial agent ordering, which was already suffering from a relatively flawed completion rate.

However, the scenario with Rare breakdowns affected the completion rate for plans across all of our ordering approaches. We highlight the fact that the completion rates of the Flatland initial ordering as well as of the Fast-First and Close-First are

Figure 5.15: Testing results obtained with Moderate breakdowns



(a) completion rates

(b) Sum of individual costs

(c) Breakdown occurrences count

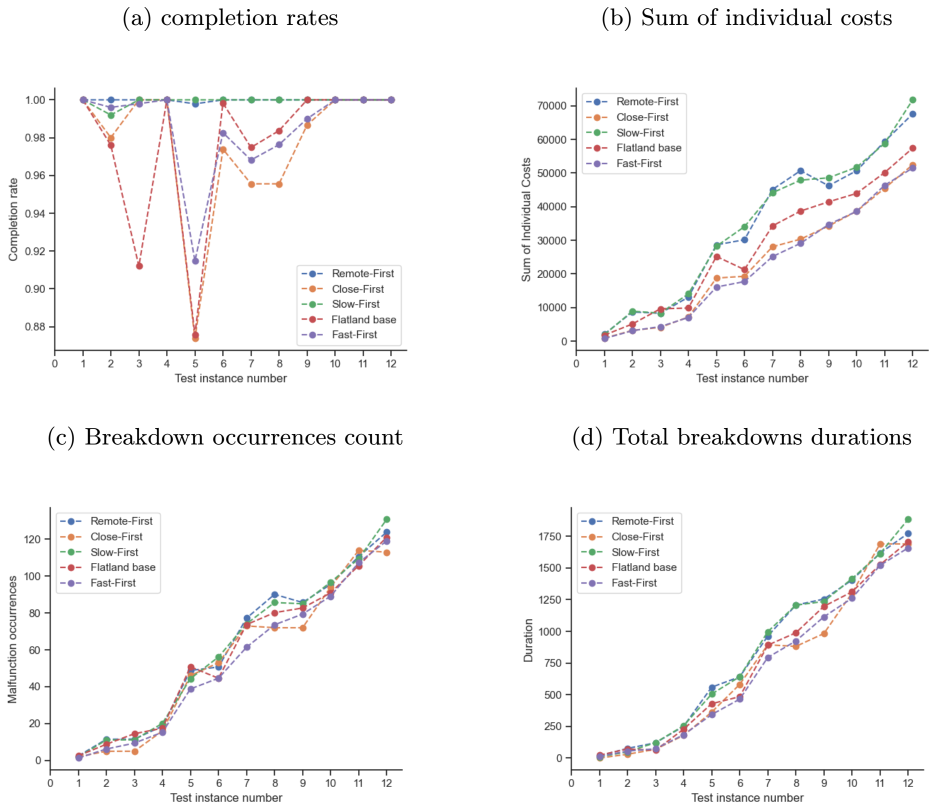
(d) Total breakdowns durations

now severely sub-par compared to other approaches. Moreover, the Remote-First and Slow-First orderings that until now had a perfect completion rate suffered slight losses of up to 1% in the second and fifth test instance. However, their performance remained robust compared to the other orderings, even in this extreme breakdown scenario.

Figures 5.14c, 5.15c and 5.16c depicting the total count of breakdown confirm that breakdown occurrences are up to 10 times more common for the scenario with Frequent breakdowns then for the scenario with Rare breakdowns. Moreover, when reviewing Figures 5.14d, 5.15d and 5.16d showing the total duration of breakdowns for our scenarios, we see that all our scenarios were exposed to similar total breakdown durations across test instances. This suggests that we succeeded in building comparable breakdown scenarios through the usage of the expectation indicator.

From Figures, 5.14c, 5.15c and 5.16c we also observe that, as expected, all plans created using the Fast-First and Close-First agent orderings suffer fewer break-

Figure 5.16: Testing results obtained with Rare breakdowns



(a) completion rates

(b) Sum of individual costs

(c) Breakdown occurrences count

(d) Total breakdowns durations

downs across all breakdown scenarios compared to the other ordering approaches. This trend is understandable as these ordering approaches aim to quickly complete agents with paths that are short from the time-step perspective. Thus they aim to minimise the chance that they will break down and block the environment. Although, the differences may seem marginal when looking at Figures depicting the breakdown occurrence counts (Figures 5.14c, 5.15c and 5.16c) and breakdown durations (Figures 5.14d, 5.15d and 5.16d), they can range up to several hundred time-steps in some test instances. For example, looking at the test instance number 8 in Figures 5.14d, 5.15d and 5.16d, we observe a difference of approximately 500 time-steps between the total breakdown durations for the Remote-First and Fast-First orderings.

Lastly, when analysing the sum of individual costs across different scenarios, we can see that the general trend remains the same, but there is an increase of up to 10,000 for the extreme cases between the scenario without any breakdowns and the scenario with Rare breakdowns. We witnessed a similar trend of increases in the make-span measurement, which we did not include, but can be found in the Figure **??** in the Appendix 6.

In overall, using the Fast-First and Close-First agent orderings resulted in significantly worse results on instances with breakdowns then using the Slow-First and Remote-First agent orderings. The sensitivity of Fast-First and Close-First orderings to breakdowns is higher because, as we explained before, some of the schedules planned using Fast-First and Close-First agent orderings were already close to their time-step limits before the occurrences of breakdowns. Therefore, when breakdowns started to occur and begin holding up lower priority agents either directly by breaking them down or indirectly through breakage of other agents that blocked their path, the completion rate started to decrease further. Understandably the decrease was the most severe for the scenario with Rare breakdowns, where a long-lasting breakdown of one agent can prevent completion for many lower priority agents.

Chapter 6

Conclusion

In the final chapter of our work, we will provide a comprehensive synthesis of our findings. Our main goals were to conduct a review of the academic literature concerned with the MAPF, to introduce the Flatland Challenge problem alongside its mathematical formulation and most importantly to propose and analyse a new heuristic approach for solving instances of the Flatland Challenge.

In the Chapter 3, we have provided a structured and comprehensive review of the most significant research on the MAPF over the last 40 years. We have covered the introduction of the MAPF as well as beginnings of the research into MAPF problems; moreover, we have reviewed the most important MAPF solving approaches and categorised them into optimal and sub-optimal, with further within category structuring based on conceptual foundations of individual approaches. Lastly, we have provided a section focused solely on introducing the literature concerned with MAPF problems where agents suffer from imperfect action executions. Our research into the literature related to the MAPF can serve as a well-rounded introduction to the problem to any newcomer to the discipline as well as a comprehensive overview for experienced researchers.

In the Chapter 2, we have thoroughly described the Flatland Challenge grid environment as well as the key mechanism regarding agent movement and agent breakdowns. Moreover, in the second part of the description, we have introduced a detailed mathematical formulation of the Flatland Challenge problem, which generalised the MAPF to the MAPF-T. Our description of the Flatland Challenge and its key mechanisms alongside their mathematical formulation allows anyone without pre-existing knowledge about Flatland to quickly understand all of its concepts as well as the crucial differences between Flatland and the standard MAPF problems covered in the contemporary literature.

In the Chapter 4 we have introduced a new MAPF-T solving approach tailored to solve instances of the Flatland Challenge. The introduced p-TPG approach is a two-stage algorithm that first plans non-conflicting paths for agents using the prioritised planning, which runs a low-level A* search with a conflict avoidance table. In the second stage, p-TPG creates the temporal plan graph based on paths planned in the first stage. The temporal plan graph controls the plan execution through dynamic precedence maintenance, which allows us to effectively handle agent breakdowns without causing conflicts. Furthermore, we have developed four new agent ordering heuristics that can be used with the prioritised planning and defined a new complexity reduction improvement for the TPG construction algorithm that significantly reduced the run-time of the algorithm.

Through the iterative testing of the p-TPG on our own test instances, we have uncovered several key learnings about the usage of it on Flatland. First of all, we have shown that the Distance map is a more suitable A* heuristics for the Flatland instances than the Manhattan distance heuristic as the planning of agents' paths using the Distance map heuristic is up to five times faster and these plans also have a higher completion rate under the Flatland time-step limit. Additionally, we have demonstrated that, on the most demanding instances, our improvement of the TPG construction algorithm results in up to twenty-five times faster TPG construction; thus significantly reducing the run-time of the p-TPG in overall.

Lastly, we have compared the performance of the p-TPG using different agent orderings heuristics. We have found that, in terms of the completion rate, agent ordering heuristics that prioritise agents with longer plan execution dominate the initial Flatland ordering as well as approaches that prioritise agents with shorter plan execution both on instances without and with breakdowns. However, we note that the differences in the completion rate across our agent ordering heuristics are a direct consequence of the time-step limit imposed by Flatland. Without the time-step limit, all agent ordering heuristics would provide an identical completion rate as conflicts never occur in our plans or during our plan execution; thus, the partial completion of any instance is caused solely by the fact that some agents cannot reach their target positions under the given time-step limit. Without the time-step limit, the orderings prioritising agents with shorter plan execution would be more suitable for Flatland as they have a lower sum-of-individual cost, shorter planning run-time due to a lower number of state expansions in the A* search, and generally lead to a less frequent occurrence of breakdowns.

Finally, we have demonstrated that the p-TPG approach paired with the Distance map state-space search heuristic and either Slow-First or Remote-First agent ordering heuristic can flawlessly, and within a reasonable time, solve even the most extreme instances of the Flatland Challenge with no breakdowns, frequent breakdowns, and moderate breakdowns. Moreover, both of these approaches fully solved all, but one instance, in the testing with the rare occurring but long breakdowns. However, we note that the breakdown configurations that we used were more demanding than configurations used in the Flatland Challenge as our configurations had five-time higher expected breakdown duration per time-step when compared to the Flatland Challenge official test cases.

The p-TPG can serve as a foundation for a further study of MAPF-T problems with very sparse grids and imperfect executions. However, the approach is not limited to it and can be used for solving standard MAPF problems. Future research into the problem of MAPF-T on sparse grids with imperfect plan execution could focus on the implementation of the CBS with the necessary improvements introduced by Boyarski *et al.* [2015], Felner *et al.* [2018], Li *et al.* [2019a], Li *et al.* [2019b] and Li *et al.* [2020]. This would allow us to replace the prioritised planning in the first stage of the p-TPG; thus, we would be planning optimal paths for agents. Furthermore, an algorithm allowing for efficient partial re-planning during the execution of plans would further improve the performance and usability of the p-TPG approach.

Bibliography

Atzmon, D., R. Stern, A. Felner, N. R. Sturtevant, & S. Koenig (2020): "Probabilistic robust multi-agent path finding." .

Atzmon, D., R. Stern, A. Felner, G. Wagner, R. Barták, & N.-F. Zhou (2018): "Robust multi-agent path finding." In "Eleventh Annual Symposium on Combinatorial Search," .

Barták, R., J. Švancara, & M. Vlk (2018): "A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs." pp. 748–756.

Bnaya, Z. & A. Felner (2014): "Conflict-oriented windowed hierarchical cooperative a." In "2014 IEEE International Conference on Robotics and Automation (ICRA)," pp. 3743–3748. IEEE.

Bnaya, Z., R. Stern, A. Felner, R. Zivan, & S. Okamoto (2013): "Multi-agent path finding for self interested agents." In "Sixth Annual Symposium on Combinatorial Search," .

Botea, A. & P. Surynek (2015): "Multi-agent path finding on strongly biconnected digraphs." In "Twenty-Ninth AAAI Conference on Artificial Intelligence," .

Boyarski, E., A. Felner, R. Stern, G. Sharon, E. Shimony, O. Bezalel, & D. Tolpin (2015): "Improved conflict-based search for optimal multi-agent path finding." *IJCAI-2015* .

De Wilde, B., A. W. Ter Mors, & C. Witteveen (2014): "Push and rotate: a complete multi-agent pathfinding algorithm." *Journal of Artificial Intelligence Research* **51**: pp. 443–492.

Dijkstra, E. W. (1959): "A note on two problems in connexion with graphs." *NUMERISCHE MATHEMATIK* **1(1)**: pp. 269–271.

Erdem, E., D. G. Kisa, U. Oztok, & P. Schüller (2013): "A general formal framework for pathfinding problems with multiple agents." In "Twenty-Seventh AAAI Conference on Artificial Intelligence," .

Erdmann, M. & T. Lozano-Pérez (1987): "On multiple moving objects." *Algorithmica* **2(1-4)**: pp. 477–521.

Felner, A., M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. Sturtevant, J. Schaeffer, & R. Holte (2012): "Partial-expansion a* with selective node generation." In "Twenty-Sixth AAAI Conference on Artificial Intelligence," .

Felner, A., J. Li, E. Boyarski, H. Ma, L. Cohen, T. S. Kumar, & S. Koenig (2018): "Adding heuristics to conflict-based search for multi-agent path finding." In "Twenty-Eighth International Conference on Automated Planning and Scheduling," .

Felner, A., R. Stern, S. E. Shimony, E. Boyarski, M. Goldenberg, G. Sharon, N. Sturtevant, G. Wagner, & P. Surynek (2017): "Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges." In "Tenth Annual Symposium on Combinatorial Search," .

Goldenberg, M., A. Felner, R. Stern, G. Sharon, & J. Schaeffer (2012): "A* variants for optimal multi-agent pathfinding." In "Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence," .

Goldenberg, M., A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. C. Holte, & J. Schaeffer (2014): "Enhanced partial expansion a." *Journal of Artificial Intelligence Research* **50**: pp. 141–187.

Hart, P. E., N. J. Nilsson, & B. Raphael (1968): "A formal basis for the heuristic determination of minimum cost paths." *IEEE Transactions on Systems Science and Cybernetics* **SSC-4(2)**: pp. 100–107.

Hönig, W., S. Kiesel, A. Tinka, J. W. Durham, & N. Ayanian (2019): "Persistent and robust execution of mapf schedules in warehouses." *IEEE Robotics and Automation Letters* **4(2)**: pp. 1125–1131.

Hönig, W., T. K. S. Kumar, L. Cohen, H. Ma, H. Xu, N. Ayanian, & S. Koenig (2017): "Summary: Multi-agent path finding with kinematic constraints." pp. 4869–4873.

Khorshid, M. M., R. C. Holte, & N. R. Sturtevant (2011): "A polynomial-time algorithm for non-optimal multi-agent pathfinding." In "Fourth Annual Symposium on Combinatorial Search," .

Li, J., A. Felner, E. Boyarski, H. Ma, & S. Koenig (2019a): "Improved heuristics for multi-agent path finding with conflict-based search." In "Proceedings of the 28th International Joint Conference on Artificial Intelligence," pp. 442–449. AAAI Press.

Li, J., G. Gange, D. Harabor, P. J. Stuckey, H. Ma, S. Koenig, J. Li, K. Sun, H. Ma, A. Felner *et al.* (2020): "New techniques for pairwise symmetry breaking in multi-agent path finding." In "International Conference on Automated Planning and Scheduling," pp. 6087–6095.

Li, J., D. Harabor, P. J. Stuckey, H. Ma, & S. Koenig (2019b): "Symmetry-breaking constraints for grid-based multi-agent path finding." In "Proceedings of the AAAI Conference on Artificial Intelligence," volume 33, pp. 6087–6095.

Luna, R. & K. E. Bekris (2011): "Efficient and complete centralized multi-robot path planning." In "2011 IEEE/RSJ International Conference on Intelligent Robots and Systems," pp. 3268–3275. IEEE.

Ma, H., D. Harabor, P. J. Stuckey, J. Li, & S. Koenig (2019): "Searching with consistent prioritization for multi-agent path finding." In "Proceedings of the AAAI Conference on Artificial Intelligence," volume 33, pp. 7643–7650.

Ma, H., S. Koenig, N. Ayanian, L. Cohen, W. Hoenig, T. K. Satish Kumar, T. Uras, H. Xu, C. Tovey, & G. Sharon (2017): "Overview: Generalizations of Multi-Agent Path Finding to Real-World Scenarios." *arXiv e-prints* arXiv:1702.05515.

Ma, H., T. S. Kumar, & S. Koenig (2017): "Multi-agent path finding with delay probabilities." In "Thirty-First AAAI Conference on Artificial Intelligence," .

Ma, H., G. Wagner, A. Felner, J. Li, T. Kumar, & S. Koenig (2018): "Multi-agent path finding with deadlines." *arXiv preprint arXiv:1806.04216* .

Morris, R., C. S. Pasareanu, K. Luckow, W. Malik, H. Ma, T. S. Kumar, & S. Koenig (2016): "Planning, scheduling and monitoring for airport surface operations." In "Workshops at the Thirtieth AAAI Conference on Artificial Intelligence," .

Nilson, N. (1980): "Principles of artificial intelligence, palo alto, calif."

Russell, S. & P. Norvig (2009): *Artificial Intelligence: A Modern Approach*. USA: Prentice Hall Press, 3rd edition.

Ryan, M. (2008): "Exploiting subgraph structure in multi-robot path planning." *Journal of Artificial Intelligence Research* **31**: pp. 497–542.

Sajid, Q., R. Luna, & K. E. Bekris (2012): "Multi-agent pathfinding with simultaneous execution of single-agent primitives." In "SoCS," .

SBB, AI Crowd (2020): "Flatland challenge." `https://www.aicrowd.com/challenges/flatland-challenge`, [Accessed: 5 January 2020].

Sharon, G., R. Stern, A. Felner, & N. R. Sturtevant (2015a): "Conflict-based search for optimal multi-agent pathfinding." *Artificial Intelligence* **219**: pp. 40–66.

Sharon, G., R. Stern, A. Felner, & N. R. Sturtevant (2015b): "Conflict-based search for optimal multi-agent pathfinding." *Artif. Intell.* **219(C)**: pp. 40–66.

Sharon, G., R. Stern, M. Goldenberg, & A. Felner (2013): "The increasing cost tree search for optimal multi-agent pathfinding." *Artificial Intelligence* **195**: pp. 470–495.

Silver, D. (2005): "Cooperative pathfinding." pp. 117–122.

Standley, T. (2010a): "Finding optimal solutions to cooperative pathfinding problems." volume 1.

Standley, T. (2010b): "Finding optimal solutions to cooperative pathfinding problems." *Proceedings of the National Conference on Artificial Intelligence* **1**.

Stern, R., N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. Satish Kumar, E. Boyarski, & R. Bartak (2019): "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks." *arXiv e-prints* arXiv:1906.08291.

Stout, B. (1998): "Smart moves: Intelligent pathfinding."

Surynek, P. (2009): "A novel approach to path planning for multiple robots in bi-connected graphs." In "2009 IEEE International Conference on Robotics and Automation," pp. 3613–3619. IEEE.

Surynek, P. (2010): "An optimization variant of multi-robot path planning is intractable." In "Twenty-Fourth AAAI Conference on Artificial Intelligence," .

Surynek, P. (2012): "A sat-based approach to cooperative path-finding using all-different constraints." In "SOCS," .

Svancara, J. & P. Surynek (2017): "New flow-based heuristic for search algorithms solving multi-agent path finding." In "ICAART (2)," pp. 451–458.

Veloso, M. M., J. Biswas, B. Coltin, & S. Rosenthal (2015): "Cobots: Robust symbiotic autonomous mobile service robots." In "IJCAI," p. 4423.

Wang, K.-H. C. & A. Botea (2011): "Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees." *Journal of Artificial Intelligence Research* **42**: pp. 55–90.

Yoshizumi, T., T. Miura, & T. Ishida (2000): "A* with partial expansion for large branching factor problems." In "AAAI/IAAI," pp. 923–929.

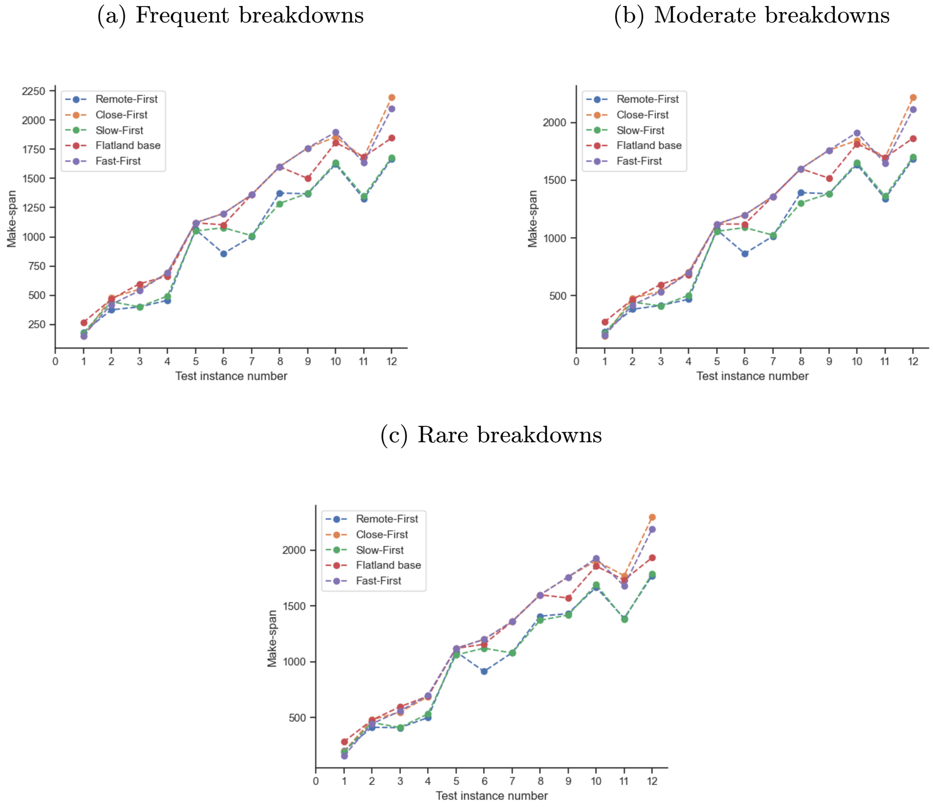
Yu, J. & S. M. LaValle (2013): "Planning optimal paths for multiple robots on graphs." In "2013 IEEE International Conference on Robotics and Automation," pp. 3612–3617. IEEE.

Zelinsky, A. (1992): "A mobile robot exploration algorithm." *IEEE Transactions on Robotics and Automation* **8(6)**: pp. 707–717.

Appendix A

In this appendix, we attach the additional test results that were not included in the Chapter 5.

Make-span values across testing with breakdowns



(a) Frequent breakdowns

(b) Moderate breakdowns

(c) Rare breakdowns

Appendix B

In this Appendix we explain how readers can install the Flatland Challenge environment and also run our pre-prepared examples on their machines.

Installing required packages

Installing all requirements for the Flatland Challenge is relatively easy and can be done in few steps, which are as follows:

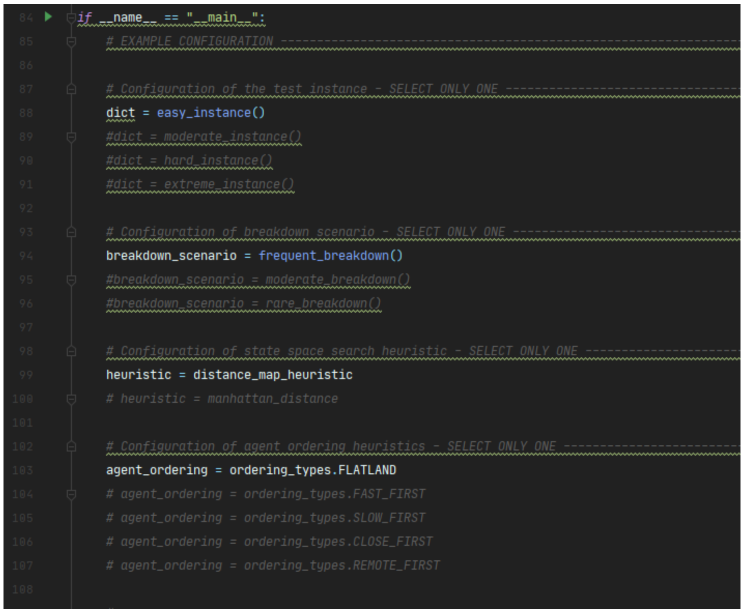
1. Install Anaconda by following instructions at Anaconda website
2. Create and activate a new conda environment named flatland-rl:
    - conda create python=3.6 –name flatland-rl
    - conda activate flatland-rl
3. Use pip to install the flatland back-end via:
    - pip install flatland-rl
4. Install additional dependencies required for our implementation:
    - conda install -c anaconda networkx
    - conda install -c anaconda seaborn

Running a Flatland example

We provide readers with a simple script allowing them to run several test cases of the Flatland themselves. Moreover, we note that the example script is located at: ”ryzner_thesis/code/Examples/example_run”.

The Figure below shows the part of the example script, where the user can configure the example instance. Lines 106 to 109 correspond to the test case selection, lines 112 to 114 correspond to the breakdown scenario selection, lines 117 and 118 determine the choice of state space search heuristic function, and finally lines 121 to 125 determine the selection of the agent ordering heuristic. Provided test cases as well as configuration options directly correspond to instances and configurations that we used to obtain our test results.

Configuration of the example

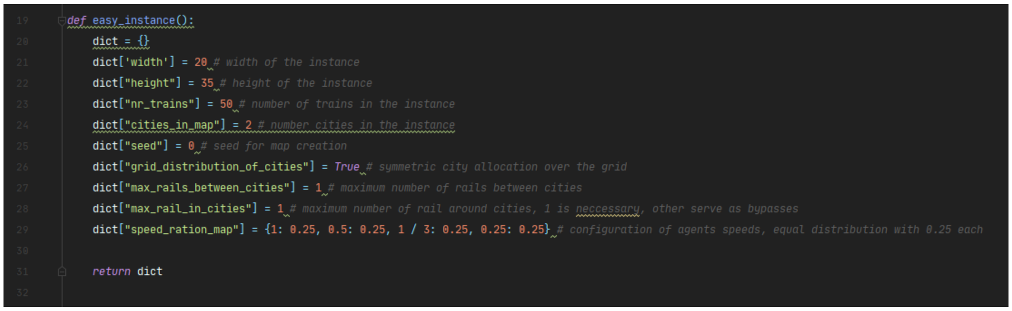


```
 84  if __name__ == "__main__":
 85      # EXAMPLE CONFIGURATION -------------------------------------------------------------
 86
 87      # Configuration of the test instance - SELECT ONLY ONE ------------------------------
 88      dict = easy_instance()
 89      #dict = moderate_instance()
 90      #dict = hard_instance()
 91      #dict = extreme_instance()
 92
 93      # Configuration of breakdown scenario - SELECT ONLY ONE -----------------------------
 94      breakdown_scenario = frequent_breakdown()
 95      #breakdown_scenario = moderate_breakdown()
 96      #breakdown_scenario = rare_breakdown()
 97
 98      # Configuration of state space search heuristic - SELECT ONLY ONE -------------------
 99      heuristic = distance_map_heuristic
100      # heuristic = manhattan_distance
101
102      # Configuration of agent ordering heuristics - SELECT ONLY ONE ----------------------
103      agent_ordering = ordering_types.FLATLAND
104      # agent_ordering = ordering_types.FAST_FIRST
105      # agent_ordering = ordering_types.SLOW_FIRST
106      # agent_ordering = ordering_types.CLOSE_FIRST
107      # agent_ordering = ordering_types.REMOTE_FIRST
108
```

We do not provide any generator of random instances to keep the example short and simple; however, readers can simply modify one of definitions of provided test instances as depicted in the Figure below.

Finally, when running the code, we provide printouts about the state of the planning, the phase of the execution and at the end we provide a brief summary of the instance results as shown in the Figure below.

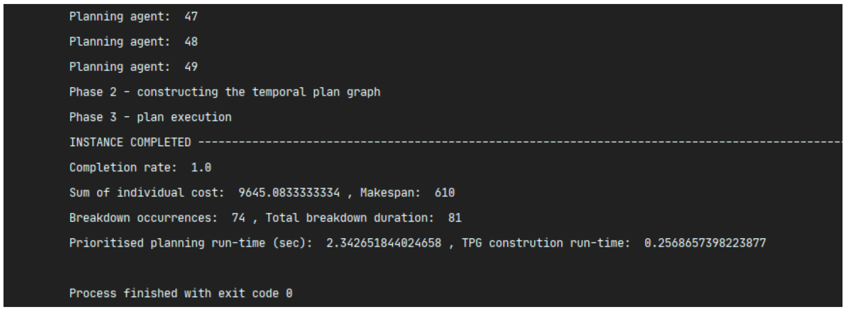
Manual instance creation



```python
def easy_instance():
    dict = {}
    dict['width'] = 20 # width of the instance
    dict["height"] = 35 # height of the instance
    dict["nr_trains"] = 50 # number of trains in the instance
    dict["cities_in_map"] = 2 # number cities in the instance
    dict["seed"] = 0 # seed for map creation
    dict["grid_distribution_of_cities"] = True # symmetric city allocation over the grid
    dict["max_rails_between_cities"] = 1 # maximum number of rails between cities
    dict["max_rail_in_cities"] = 1 # maximum number of rail around cities, 1 is neccessary, other serve as bypasses
    dict["speed_ration_map"] = {1: 0.25, 0.5: 0.25, 1 / 3: 0.25, 0.25: 0.25} # configuration of agents speeds, equal distribution with 0.25 each

    return dict
```

Run-time information printout



```
Planning agent:  47
Planning agent:  48
Planning agent:  49
Phase 2 - constructing the temporal plan graph
Phase 3 - plan execution
INSTANCE COMPLETED ---------------------------------------------------------------------------------------------
Completion rate:  1.0
Sum of individual cost:  9645.0833333334 , Makespan:  610
Breakdown occurrences:  74 , Total breakdown duration:  81
Prioritised planning run-time (sec):  2.342651844024658 , TPG constrution run-time:  0.2568657398223877


Process finished with exit code 0
```

Attached videos

Finally, we also provide videos of 3 example instances, which can be found in the following folder: ”ryzner_thesis/video”. Moreover, the reader can use the following link to download videos from the online drive: repository

Appendix C

Alongside our work, we provide a DVD with the following structure:

```
Ryzner_thesis
    Codes
        Examples
        Thesis_code
    Media
    Thesis_text
```

where the folder *Codes* contains a run-able example in the sub-folder *Examples* and the full solution code in the sub-folder *Thesis_code.* The folder *Media* contains 3 video examples from the run-time of the Flatland simulation, and finally the folder *Thesis_text* contains the file with our work in the *pdf* format.