Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)

# Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge

**Jiaoyang Li,[1*] Zhe Chen,[2] Yi Zheng,[1] Shao-Hung Chan,[1]**
**Daniel Harabor,[2] Peter J. Stuckey,[2] Hang Ma,[3] Sven Koenig[1]**
[1]University of Southern California, USA
[2]Monash University, Australia
[3]Simon Fraser University, Canada
{jiaoyanl, yzheng63, shaohung, skoenig}@usc.edu, {zhe.chen, daniel.harabor, peter.stuckey}@monash.edu, hangma@sfu.ca

**Abstract**

Multi-Agent Path Finding (MAPF) is the combinatorial problem of finding collision-free paths for multiple agents on a graph. This paper describes MAPF-based software for solving train planning and replanning problems on large-scale rail networks under uncertainty. The software recently won the 2020 Flatland Challenge, a NeurIPS competition trying to determine how to efficiently manage dense traffic on rail networks. The software incorporates many state-of-the-art MAPF or, in general, optimization technologies, such as prioritized planning, large neighborhood search, safe interval path planning, minimum communication policies, parallel computing, and simulated annealing. It can plan collision-free paths for thousands of trains within a few minutes and deliver deadlock-free actions in real-time during execution.

## 1 Introduction

The Flatland Challenge (Mohanty et al. 2020) is a competition set up to answer the question "How to efficiently manage dense traffic on complex rail networks?" It is organized by AIcrowd, an online AI crowd sourcing platform, and Swiss Federal Railways, Deutsche Bahn, and SNCF, three large railway network operators. The first iteration of the competition happened in 2019. The second iteration happened in 2020 and is what this paper discusses. The Flatland Challenge consists of an idealized rail planning problem. Given a map showing rail tracks and train stations (see Figure 1) and a set of trains with start and target stations, the task is to design a plan so that each train moves from its start station to its target station within a given time limit, while respecting the usage of track segments, i.e., no two trains occupy the same track segment (a vertex collision) or cross each other by moving in opposite directions from adjacent track segments (an edge collision) at the same time.

The academic version of the Flatland Challenge is called Multi-Agent Path Finding (MAPF), which is moving multiple agents from start to target locations on a graph while avoiding vertex and edge collisions. MAPF is the quintessential movement coordination problem and is widely applicable in computer games (Sigurdson et al. 2018; Li et al. 2020b), automated valet parking (Okoso, Otaki, and Nishi 2019), UAV traffic management (Ho et al. 2019), drone swarms (Hönig et al. 2018), and automated warehousing (Ma et al. 2017; Li et al. 2021b).

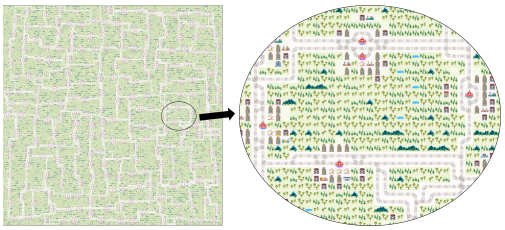


Figure 1: Flatland environment represented by a $229 \times 229$ grid map. The grey lines represent rail tracks, clusters of buildings represent cities, and the red houses on the rail tracks represent (some of the) train stations.

The Flatland Challenge is an exciting event and has been proved very attractive to enter. The 2020 Flatland Challenge involved more than 700 participants from 51 countries making more than 2,000 submissions over 4 months; 64 teams participated in Round 1 and 44 teams in Round 2. In this paper, we explain the methods we tried out in tackling the Flatland Challenge, and some of the peculiarities that arose in the challenge itself; culminating in our final approach, MAPF-based software that won both rounds of the 2020 Flatland Challenge. The software incorporates many state-of-the-art MAPF or, in general, optimization technologies, such as prioritized planning (Silver 2005), large neighborhood search (Shaw 1998), safe interval path planning (Phillips and Likhachev 2011), minimum communication policies (Ma, Kumar, and Koenig 2017), parallel computing, and simulated annealing. It can plan collision-free paths for thousands of trains within a few minutes and deliver deadlock-free actions in real-time during execution.

## 2 Flatland Environment

In this section, we introduce the Flatland environment and explain its relationship to MAPF.

### 2.1 Problem Definition

A Flatland environment simulates a rail network and its trains in a simplified way. The rail network is represented by

[*]Jiaoyang Li performed her research during her visit to Monash University.


Figure 2: Eight rail types: (a) straight, (b) curve, (c) simple switch, (d) diamond crossing, (e) single slip switch, (f) double slip switch, (g) tri-symmetrical switch, and (h) symmetrical switch. Each rail type can be rotated or reflected.

a $w \times h$ grid map with $n$ cities. Each unblocked cell is associated with a *rail type*, as shown in Figure 2, that determines how the train can move through the cell. Each city contains 2, 3, or 4 parallel rail tracks, and each rail track in a city contains a train station. There are $m$ trains $a_1, a_2, ..., a_m$, each with a start cell (= a train station), an initial orientation, and a target cell (= another train station). Time is discretized into timesteps from 0 to $T_{max} = \lfloor 8(w + h + \frac{m}{n}) \rfloor$. Our task is to give commands to the trains at every timestep so that we move as many trains as possible to their target cells as early as possible. To be more specific, we want to maximize the *normalized reward* (or *reward* for short) $\frac{\delta}{T_{max}} - \frac{\sum_{1 \leq i \leq m} T_i}{mT_{max}} \in [-1, 0]$, where $T_i \in [1, T_{max}]$ is the *arrival time* of train $a_i$ at its target cell if it reaches its target cell by timestep $T_{max}$ and $T_{max}$ otherwise, and $\delta$ is 1 if all trains reach their target cells by timestep $T_{max}$ and 0 otherwise. We define the *success rate* as the percentage of trains that reach their target cells by timestep $T_{max}$. If the success rate of a given instance is 1, then maximizing the reward is equivalent to minimizing the *flowtime* $\sum_{1 \leq i \leq m} T_i$, which is a common metric to evaluate the solution quality in the MAPF literature. We define the *earliest arrival time* $T_i^0$ of train $a_i$ as the earliest timestep when it can reach its target cell when ignoring collisions with other agents.

At timestep 0, no trains are in the environment. For each train, we determine its *departure time* and give a command at that time to let it enter the environment, i.e., appear at its start cell with its initial orientation, which takes one timestep. While the train is in the environment, it always occupies one cell at each timestep. We give a move or stop command to it at every timestep until it reaches its target cell. The train immediately leaves the environment when it reaches its target cell. If we give a stop command, the train stops and waits at its current cell. If we give a move forward/left/right command (following the transition rule indicated by the rail type of its current cell), then the train needs one timestep to move to the corresponding adjacent cell iff (1) this move action does not conflict with the actions of any other trains; and (2) the train is not suffering from a malfunction. It remains at its current cell otherwise. We say two actions conflict iff they lead to a collision. The definition of collisions was slightly different in Round 1 and Round 2 and thus will be introduced in detail in the corresponding sections (i.e., Sections 3.4 and 4.1). For clarity, we assume for now that two trains *collide* iff they are at the same cell or traverse two adjacent cells in opposite directions at the same timestep. *Malfunctions* simulate delays by stopping a train at a random timestep for a random duration, where, for each train, the random timestep is generated by a Poisson process with a known rate $\lambda$ (called *malfunction rate*) and is not known a priori, and the random duration is uniformly selected between 20 and 50 timesteps and becomes known when the malfunction occurs.

A software library written in Python was provided to simulate the environment. The simulation on one instance finishes when all trains reach their target cells or $T_{max}$ is reached. The simulation on the next instance starts only after the simulation on the previous instance has finished. To save on runtime, our solution software[1] is written as a dynamic library in C++ and called by the Python simulator.

### 2.2 Relationship with MAPF and Its Variants

MAPF is a broad family of problems with many variants (Stern et al. 2019). A widely used formulation defines MAPF on an undirected graph with a team of agents, each with a start and target vertex. At each timestep, an agent can either move to an adjacent vertex or wait at its current vertex. Agents are at their start vertices at timestep 0 and remain at their target vertices after they complete their paths (where they can still collide with other agents). The task is to move all agents with minimum flowtime to their target vertices without collisions.

The Flatland Challenge has important differences to standard MAPF but is related to some MAPF variants:

1. Train movement is restricted to rails, which make up a small proportion of the map. Trains may not move backward. This requires us to take into account the orientation of the trains when planing paths, which is related to MAPF with motion planning (Cohen et al. 2019).
2. Trains enter the environment over time and leave it after reaching their target cells, which is related to online MAPF (Svancara et al. 2019).
3. We are asked to move as many trains as possible (instead of all trains) to their target cells before a given timestep, which is related to MAPF with deadlines (Ma et al. 2018).
4. Trains break down randomly while moving and remain then stationary at the breakdown location for a number of timesteps, which is related to MAPF with delay probabilities or stochastic travel times (Cáp, Gregoire, and Frazzoli 2016; Ma, Kumar, and Koenig 2017; Wagner and Choset 2017; Li et al. 2019; Coskun and O'Kane 2019; Atzmon et al. 2020; Street et al. 2020).

## 3 Round 1

Round 1 required solving 400 instances with 14 different settings (see Table 1) in a random order within 8 hours (including both planning and simulation time) while maximizing the mean reward. The reward for each unsolved instance is -1. For each instance, we were given up to 300 seconds to generate the first action commands at timestep 0 (i.e., before *execution*) and then up to 5 seconds to generate the next action commands at every timestep (i.e., during *execution*).

[1] https://github.com/Jiaoyang-Li/Flatland

| test | $m$ | $w$ | $h$ | $n$ | $\lambda$ | #instances |
|---|---|---|---|---|---|---|
| Test_0 | 5 | 25 | 25 | $\leq 2$ | $\leq 1/50$ | 50 |
| Test_1 | 10 | 30 | 30 | $\leq 2$ | $\leq 1/100$ | 50 |
| Test_2 | 20 | 30 | 30 | $\leq 3$ | $\leq 1/200$ | 50 |
| Test_3 | 50 | 20 | 35 | $\leq 3$ | $\leq 1/500$ | 40 |
| Test_4 | 80 | 35 | 20 | $\leq 5$ | $\leq 1/800$ | 30 |
| Test_5 | 80 | 35 | 35 | $\leq 5$ | $\leq 1/800$ | 30 |
| Test_6 | 80 | 40 | 60 | $\leq 9$ | $\leq 1/800$ | 30 |
| Test_7 | 80 | 60 | 40 | $\leq 13$ | $\leq 1/800$ | 30 |
| Test_8 | 80 | 60 | 60 | $\leq 17$ | $\leq 1/800$ | 20 |
| Test_9 | 100 | 80 | 120 | $\leq 21$ | $\leq 1/1000$ | 20 |
| Test_10 | 100 | 100 | 80 | $\leq 25$ | $\leq 1/1000$ | 20 |
| Test_11 | 200 | 100 | 100 | $\leq 29$ | $\leq 1/2000$ | 10 |
| Test_12 | 200 | 150 | 150 | $\leq 33$ | $\leq 1/2000$ | 10 |
| Test_13 | 400 | 150 | 150 | $\leq 37$ | $\leq 1/4000$ | 10 |

Table 1: The settings of the 400 instances used in Round 1.

### 3.1 Finding the Approach to Path Compatibility

The Flatland Challenge is a MAPF problem at its core. The first three differences discussed in Section 2.2 can be addressed with small modifications to existing MAPF algorithms. The last difference, namely the malfunctions, can be handled during execution. We therefore started with trying variants of existing MAPF algorithms to plan collision-free paths under the assumption that no malfunctions occur.

**Conflict Based Search (CBS) with Corridor Reasoning** CBS (Sharon et al. 2015) is a leading approach to solving MAPF optimally. It calls *space-time A** (Silver 2005), a variant of A* that finds the shortest path with respect to given spatio-temporal constraints, to plan paths for single agents on the low level and resolves collisions among the planned paths on the high level. Li et al. (2020a) demonstrate that corridor reasoning speeds up CBS significantly when the map contains corridors (i.e., chains of vertices, each of degree 2). The Flatland maps are full of single-track corridors, and the main target of Round 1 was to optimize solution quality. This makes CBS with corridor reasoning seemingly a good fit for the task, as it solves the problem optimally and is efficient on maps with corridors. We therefore tried it first. We modified space-time A* by adding the orientation of the train as an additional attribute of the state and assigning a dummy state to the root A* node to represent the train not being in the environment. However, the situation where many trains share one start cell causes CBS to exhaustively explore all possible departure sequences for the trains. In addition, corridor reasoning cannot efficiently address the situation when many trains traverse the same single-track corridor as it reasons about only two agents at a time. As a result, CBS struggles to solve instances with dozens of trains within an acceptable runtime.

**Prioritized Planning** The failure of CBS led to the idea of reducing the number of trains in each CBS search. The result was Grouped Prioritized Planning, which first sorts all trains in a priority ordering and then, from the highest priority to the lowest priority, plans paths for a group of trains using CBS while avoiding collisions with the already planned paths of all higher-priority trains. However, we abandoned CBS when we noticed that setting the group size to 1, which makes the algorithm degenerate to a vanilla *Prioritized Planning* (PP) (Silver 2005), results in the most efficient setting. In addition, settings with group sizes larger than 1 do not bring advantages in solution quality and dramatically increase the runtime. Another reason for considering PP is that the Flatland environment is "well-formed" (Ma et al. 2019), and PP with any priority ordering is thus guaranteed to find collision-free paths for all trains although not necessarily of lengths smaller than $T_{max}$. In practice, however, the path lengths generated by PP were always smaller than $T_{max}$.

### 3.2 Improving the Solution Quality

Although PP can find collision-free paths rapidly, its solution quality is far from optimal. Furthermore, the Flatland Challenge allowed 300 seconds for generating the initial plan, and PP is able to solve each Round 1 instance without reaching this runtime limit. Therefore, we wanted to use the remaining time to improve the solution quality.

**Large Neighborhood Search (LNS)** LNS (Shaw 1998) was originally invented for vehicle routing problems and is popular in solving challenging discrete optimization problems. Our recent work (Li et al. 2021a) shows the effectiveness of LNS for solving MAPF. We followed this work by using PP to generate an initial solution and repeating a neighborhood search process to improve the solution quality until the time/iteration limit is reached. In each iteration, we select a subset of $m_s$ trains ($m_s \leq m$, we used $m_s = 5$), called a *neighborhood*, and replan their paths using PP (with a random priority ordering). The new paths need to avoid collisions with each other and with the paths of all other trains. We adopt the new paths if they reduce the flowtime. We designed three neighborhood selection strategies (the first two are from (Li et al. 2021a)): (1) the train-based strategy, which selects a train $a_i$ with the largest delay $T_i - T_i^0$ and $m_s - 1$ trains that prevent train $a_i$ from reaching its target cell earlier; (2) the intersection-based strategy, which selects $m_s$ trains that visit the same *intersection* (i.e., cell of rail types (c) to (g) in Figure 2); and (3) the start-based strategy, which selects $m_s$ trains with the same start cell. We use adaptive LNS (Ropke and Pisinger 2006), which records the relative improvement of the flowtime of the three strategies and chooses strategies with probabilities proportional to its relative improvement to generate the next neighborhood.

**Parallel LNS** As the Flatland Challenge provided 4 CPUs for evaluation, we run 4 LNS threads in parallel, one for each of the following priority orderings (used by PP to generate the initial solution):

1. in order of the train indices: lowest to highest,
2. in order of the earliest arrival time: highest to lowest,
3. in order of the earliest arrival time: lowest to highest, and
4. preferring different start cells, breaking ties by maximizing the earliest arrival time. That is, we divide the agents into groups according to their start cells and iterate over the groups, each time selecting the train $a_i$ that has the smallest earliest arrival time $T_i^0$ among the trains remaining in the group and then moving to the next group.

We are interested in the earliest arrival times $T_i^0$ of the trains for two reasons. On the one hand, prioritizing trains with larger $T_i^0$ tends to increase the number of the trains that reach their target cells before timestep $T_{max}$. On the other hand, prioritizing trains with smaller $T_i^0$ tends to allow the trains with small $T_i^0$ to reach their target cells very fast and reduce the influence of their malfunctions to other trains, which in turn reduces the flowtime. In addition, the motivation behind the first priority is explained in Section 3.4, and the motivation behind fourth priority ordering is to generate an initial solution that is likely to be very different from the solutions generated by the other three priority orderings. Empirically, the third priority ordering performs better than the other ones in many cases, but there is no single priority ordering that dominates the other ones.

Since we needed to solve 400 instances within 8 hours (= 28,800 seconds), we set the runtime limit of LNS for each instance to $\max\{280, \frac{t}{l} + 5m\}$ seconds, where $t \in [0, 28800]$ is the remaining runtime budget, $l$ is the remaining number of instances, and $m$ is the number of trains. The term $5m$ allocates more runtime budget to harder instances. In order to avoid spending too many iterations on easy instances, we also set an iteration limit of 10,000 for each instance. We terminate an LNS thread when the runtime limit or the iteration limit is reached. Once all LNS threads finished, we pick the solution with the smallest flowtime. Since the instances were evaluated in a random order and we usually do not reach the runtime limit for easy instances, we always successfully solved 400 instances within 8 hours.

We tested the algorithms on our server, i.e., Nectar Research Cloud using an instance with an AMD Opteron 63xx class CPU and 64 GB RAM, using our self-generated instances. LNS reduced the flowtime on 312 out of 400 instances over PP, with an average reduction of 12.4%. Parallel LNS reduced the flowtime on 157 instances over LNS, with an average reduction of 11.1%. As a result, LNS improved the mean reward by 0.010 (= 3 times the difference to the team in second place) on our server. Parallel LNS further improved it by 0.001.

### 3.3 Recovering from Breakdowns

When a train malfunctions during execution, deadlocks might happen if we stick to the original paths. Figure 3 shows an example. Trains 1 and 3 want to move to target cell C, and train 2 wants to move to target cell A. In the original paths, they all move to their target cells without stopping. However, if train 1 malfunctions and stops at cell A, say, for 20 timesteps, then, after 4 timesteps, train 2 reaches cell B. There is a deadlock between trains 1 and 2, and, since trains are not allowed to move backward, neither of them can reach their target cells.

**Minimum Communication Policies (MCP)** MCP (Ma, Kumar, and Koenig 2017) avoids such deadlocks by stopping some trains to maintain the ordering with which each train visits each cell. It guarantees that all trains can reach their target cells within a finite number of timesteps. In the example shown in Figure 3, the ordering with which the trains visit cell D is 1, 2, and 3. Therefore, if train 1 malfunctions, train 2 waits at cell I until train 1 visits cell D first, which successfully avoids the deadlock.

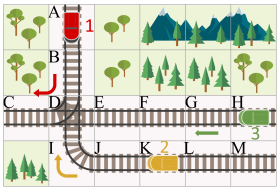



Figure 3: Three-train example.

**Partial Replanning** MCP sometimes stops trains unnecessarily. In the previous example, when train 1 malfunctions for 20 timesteps, MCP stops train 3 at cell E for 20 timesteps because train 3 has to visit cell D after train 2. However, this is unnecessary since train 3 can visit cell D first, without waiting for trains 1 or 2. Therefore, we develop a partial replanning mechanism. When train $a_i$ encounters a new malfunction at some timestep, we collect all intersections that train $a_i$ visits in the future and then collect all trains that visit at least one of these intersections after train $a_i$. We replan the paths of these trains one after the other with PP. We terminate this procedure when new paths have been planned for all these trains or the runtime limit has been reached (we used 3 seconds). Empirically, adding the partial replanning technique reduced the flowtime on 261 instances, with an average reduction of 19.9%. As a result, we improved the mean reward by 0.014 on our server.

### 3.4 Fighting with the Simulator

**Asymmetric Movement Rule** A very common rule in practice is that a train may not enter a track segment if it is not yet clear, a rule enforced by the so called absolute block signaling. MAPF with delay probabilities (Ma, Kumar, and Koenig 2017) also uses a similar movement rule, i.e., agents can move only to cells that are currently not occupied by other agents, because it guarantees that, if an agent stops unexpectedly, the agent following it will not collide with it at the next timestep. We assumed that the Flatland Challenge used the same movement rule as well, since example solutions often had two trains moving along the same rail track, separating by one unoccupied cell. However, we then realized that this behavior was not necessary but rather an artifact of the simulator. The trains in the simulator are moved one at a time in index order. Hence, if train 2 is directly in front of train 1 and has a larger index than train 1, then train 1 cannot move to the cell where train 2 still is. But if we reverse the situation, then train 2 can move to the cell where train 1 was, since train 1 moves first and leaves its cell. This means that convoys of trains can move without separation if they were ordered from lowest index in the front to highest index in the back. Once we realized this, we modified space-time A* and MCP to take this asymmetric movement rule into account, and the mean reward improved by 0.007 on our server.

**Command Commitment** Another feature of the simulator which came as a surprise to us and required quite some

workaround was the committed nature of decisions. Once a train is given a move command, this command cannot be changed until the movement succeeds. So, if train 1 tries to move to cell A but fails due to collisions or malfunctions, then we cannot update its move command to a stop command or try a different move command next. Instead, train 1 can be issued a new move or stop command only after it succeeds in moving to cell A. This issue requires a lot of effort to handle corner cases in partial replanning. For example, when we replan the path of train 1, we have to keep train 1 waiting at its current cell and immediately move it to cell A when possible. When we replan the paths of other trains, we have to prohibit them from moving to cell A at those timesteps when train 1 has not yet moved to cell A.

### 3.5 Summary

In summary, we use PP to generate an initial solution and LNS to improve its quality. During execution, we use MCP to generate action commands and partial replanning to improve the solution quality when a train encounters a new malfunction. On the leaderboard, we eventually achieved a score (= mean reward) of -0.104 with a success rate of 100%, which was 0.003 (= 3%) and 0.012 (= 10%) higher than the scores of the teams in second and third place.

## 4 Round 2

Round 2 required maximizing the accumulated (modified) reward over an infinite number of instances of increasing difficulty within 8 hours, where the reward for each instance was changed to $\frac{\delta}{T_{max}} - \frac{\sum_{1 \leq i \leq m} T_i}{mT_{max}} + 1 \in [0, 1]$, i.e., the original reward plus one. So, there is a trade-off between spending more runtime on each instance to obtain a higher reward per instance and spending less runtime on each instance to obtain rewards from more instances.

The instances were grouped into tests of increasing difficulty. Each test contains 10 levels with different malfunction rates, where Level_0 does not have malfunctions ($\lambda = 0$), and Level_$j$ ($1 \leq j \leq 9$) has malfunction rate $\lambda = \frac{1}{250j}$. Each level contains one instance. The number of trains in each instance in Test_0 is $m_0 = 1$, and that in Test_$(i+1)$ ($i \geq 0$) is $m_{i+1} = m_i + \lceil 0.75 \times 10^{len(m_i)-1} \rceil$, where $len(x)$ is the number of digits in integer $x$. For each instance in Test_$i$ ($i \geq 0$), the number of cities is $n_i = \lfloor \frac{m_i}{10} \rfloor + 2$, and the size of the grid map is $w_i = h_i = \lceil \sqrt{150 n_i} \rceil + 7$. For instance, $m_{10} = 18, n_{10} = 3$, and $w_{10} = h_{10} = 29$; $m_{20} = 98, n_{20} = 11$, and $w_{20} = h_{20} = 48$; and $m_{30} = 781, n_{30} = 80$, and $w_{30} = h_{30} = 117$. The map shown in Figure 1 is generated using the parameters of Test_36. For each instance, we are given up to 600 seconds to generate the first action commands at timestep 0 and then up to 10 seconds to generate the next action commands at every timestep. Generally speaking, Round 2 instances were substantially more challenging than Round 1 instances. They have more trains and higher malfunction rates. For example, the most challenging instance we solved in our best submission (i.e., Test_36 Level_1) in Round 2 had 3,256 trains in total and, on average, 11.35 trains encountering new malfunctions at every timestep.

### 4.1 Fighting with the Simulator Again

**Symmetric Movement Rule** The asymmetric movement rule explained in Section 3.4 was removed in Round 2. All trains now essentially moved simultaneously in the simulator, so train 1 can move to the cell that train 2 vacates at the same timestep independent of the indices of the trains. That is, collisions were now defined exactly as in Section 2. We therefore modified space-time A* and MCP accordingly. In addition, since the movement rule was now independent of the indices of the trains, we replaced the first priority ordering for PP introduced in Section 3.2 with a priority ordering that prefers different start cells and breaks ties by minimizing the earliest arrival time.

**Observation Builder** The simulator returns an observation at every timestep, which is designed to be used by reinforcement learning algorithms to index their policies for action selection. By default, a global observation is returned at every timestep, which contains $m$ copies of the map and the train information, one for each train. Competition entries can overwrite the default and create arbitrary observations. Since we did not make use of the observation, we ignored this part of the code. Only in Round 2 did we realize that the construction of the global observation required substantial time. When we replaced the call to return a dummy empty observation, the simulator sped up substantially, e.g., by more than 6 times for the instances in Test_27 (which contain 556 trains), and the speed-up increases with the instance size.

**Simulation Runtime** The simulator runs until all trains reach their target cells or timestep $T_{max}$ is reached. Unfortunately, even after we replaced the global observation, the simulator was still quite slow. Even when no trains are moving, it still takes quite a bit of time to run. This means that strategies that try to solve more instances by planning paths for only a portion of the trains are expensive, simply in terms of simulation time. For example, when we ran the best version of our software on our server, only 30% of the 8-hour runtime budget was spent on our C++ software, and 70% was spent on the Python simulator.

### 4.2 Trade off between Runtime and Reward

There is a trade-off between spending more runtime on LNS and partial replanning to obtain a higher reward per instance and spending less runtime on LNS and partial replanning to solve more instances within 8 hours.

**Simulated Annealing** To allocate the runtime spent by LNS on each instance best, we use *Simulated Annealing* (SA) to choose the number of iterations $N_i$ for which we should run LNS on the instances in each Test_$i$. We first collect two sets of data averaged over a large number of instances (by running two sets of experiments, one with 0 iterations and halting after the simulation finishes and one with 5000 iterations and halting after LNS finishes): (1) the runtime $t^0_{ij}$ and the reward $r^0_{ij}$ per instance in each Level_$j$ Test_$i$ with 0 iterations, and (2) the runtime $\tau_{ik}$ spent by LNS and the reward improvement ratio $p_{ik}$ by LNS relative to the initial solution per instance in each Test_$i$ with $k \in [1, 5000]$ iterations (i.e, $\tau_{ik}$ is the runtime of the first $k$ LNS iterations,

and $p_{ik}$ is the flowtime of the solution at iteration $k$ divided by the flowtime of the solution at iteration 0). That is, $t^0_{ij}$ is an estimate of the runtime (including both planning and simulation time) we need to spend if we do not use LNS, $\tau_{ik}$ is an estimate of the additional runtime spent by LNS with $k$ iterations, $r^0_{ij}$ is an estimate of the reward without LNS, and $p_{ik}$ is an estimate of the reward improvement ratio for LNS with $k$ iterations. We define $t^0_{ij}$ and $r^0_{ij}$ for each level in each test but $\tau_{ik}$ and $p_{ik}$ only for each test because the values of $t^0_{ij}$ and $r^0_{ij}$ are different when the malfunction rates are different while $\tau_{ik}$ and $p_{ik}$ are irrelevant to the execution and thus irrelevant to the malfunction rates.

We build a SA model with the number of iterations $N_i$ being the variables and the accumulated reward being the objective. We set $N_i = 0$ for all Test_$i$ initially. In each SA iteration, we randomly select a group of variables and randomly increase or decrease their number of iterations by 5, 10, 20, 50, 100, 200, or 500. For the instance $(i, j)$ of Level_$j$ Test_$i$, we estimate its runtime by $t_{ij} = t^0_{ij} + \tau_{iN_i}$ and its accumulated reward by $r_{ij} = a \cdot p_{iN_i} \cdot r^0_{ij}$ if $p_{iN_i} > 1$ and $r_{ij} = p_{iN_i} \cdot r^0_{ij}$ otherwise, where $a \geq 1$ is an amplification ratio (we used 1.01), indicating that improving the solution quality before execution can further improve the reward after execution, as the reward after execution considers the influence of malfunctions and the earlier the trains reach their target cells, the fewer malfunctions they encounter on average. Then, we use $t_{ij}$ to estimate the set of instances $\mathcal{I}$ that can be solved within 8 hours and $r_{ij}$ to estimate the resulting accumulated reward $\sum_{(i,j)\in\mathcal{I}} r_{ij}$.

In practice, the best SA solution allocated non-zero LNS iterations, ranging from 5 to 1,230, on Test_2 (which contains 3 trains) to Test_27 (which contains 556 trains). When we tested it on our server, LNS (without parallelization and with the third priority ordering introduced in Section 3.2) consumed 799 seconds and improved the accumulated reward by 1.137 among the instances in Test_2 to Test_27. Eventually, we solved 1 fewer instance within 8 hours and improve the accumulated reward by 0.709 (= 61% of the difference to the team in second place).

**Parallel LNS with Leader Thread** The priority ordering used by PP to find the initial solutions can have a significant influence on not only the solution quality of LNS but also its runtime. Since the third priority ordering introduced in Section 3.2 performs overall the best, we developed a parallelization of LNS that guarantees that, when terminating, its runtime and solution quality are never worse than those of LNS with the third priority ordering alone. Specifically, we use the LNS thread with the third priority ordering as the leader thread. We then run 4 LNS threads in parallel and terminate them when (1) the leader thread terminates, or (2) a non-leader thread terminates and the flowtime of its solution is smaller than the current estimate of the flowtime of the leader thread, which is the flowtime of the already planned paths plus the sum of the earliest arrival times of the trains whose paths have not been planned yet. When we compared parallel LNS with leader thread against LNS on our server, parallel LNS with leader thread consumed 9 fewer seconds and improves the accumulated reward by 0.571 among the instances in Test_2 to Test_27.

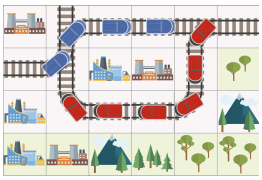


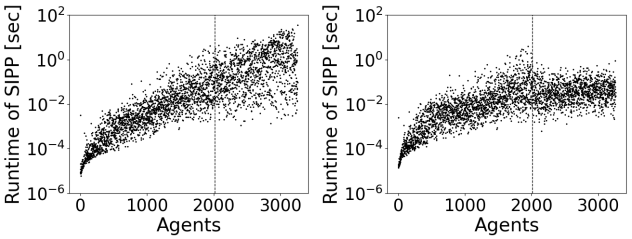
Figure 4: A traffic jam where the trains in the loop can move only when none of them malfunction.

**Partial Replanning** Partial replanning can be time-consuming for large instances since too many trains are malfunctioning and too many trains are affected by malfunctioning trains. In addition, partial replanning is less effective for large instances in terms of the reward since they have large values of $m$ and $T_{max}$. We therefore apply partial replanning only for instances in Test_1 (which contains 2 trains) to Test_28 (which contains 631 trains). On our server, partial replanning consumed 1,965 seconds on these instances and improved their accumulated reward by 4.812. Eventually, we solved 2 fewer instances within 8 hours but improved the accumulated reward by 3.629.

### 4.3 Traffic Jams

Round 2 has substantially more agents with higher malfunction rates than Round 1. We observed that malfunctions can cause severe traffic jams when too many trains are in the same region. Figure 4 shows an example: the trains in the figure form a loop and try to rotate clockwise simultaneously. However, this can be done only when none of them malfunction. Assume that there are 50 trains in the loop with a malfunction rate of $\lambda = 1/250$ (which is used for Level_1 instances) and a malfunction duration of $t_d = 35$ timesteps (which is the average malfunction duration). Assume also that $T_{max} = \infty$. If we consider only one train and ignore its collisions with other trains, then the probability of it first malfunctioning $k \in [0, \infty)$ times, which takes $kt_d$ timesteps, and then performing a move action, which takes 1 timestep, is $\lambda^k(1 - \lambda)$. So, on average, the train performs one move action during every $\sum_{k=0}^{\infty} \lambda^k(1 - \lambda)(kt_d + 1) = \frac{1-\lambda+\lambda t_d}{1-\lambda}$ timesteps and malfunctions at the remaining timesteps. That is, the probability for it not to malfunction at a given timestep is $\frac{1-\lambda}{1-\lambda+\lambda t_d}$. Thus, the probability of none of the trains in the loop malfunctioning at a given timestep is $(\frac{1-\lambda}{\lambda t_d-\lambda+1})^{50} \approx 0.14\%$, indicating that it is almost impossible for the trains to move forward (even for one timestep).

We attempted to relieve such traffic jams by taking the malfunctions into account during path planning. We modified space-time A* to find a path for a train that minimizes its expected arrival time by considering the delays caused by both the malfunctions of the train and the expected delays of the train ahead of it. Although minimizing the expected arrival times has been shown to be a successful approach in the MAPF literature (Ma, Kumar, and Koenig 2017; Li et al. 2019; Atzmon et al. 2020), it did not work for us because




(a) Without lazy planning (b) With lazy planning

Figure 5: Comparison of the runtimes of SIPP without and with lazy planning. The paths of the trains to the right of the vertical dashed line are planned during execution.

trains usually have a limited number of paths to their target cells in the Flatland environment, so they often do not have paths available that avoid congested regions. We thus abandoned this approach and leave this issue for future work.

### 4.4 Speed Is All That Matters

We use space-time A* to find the shortest path for a train that avoids collisions with the given paths of all other trains. Each state of space-time A* is a pair of a cell and a timestep. As a result, for large instances with many trains, the number of timesteps is usually large, and thus the search space of space-time A* is also large, which leads to unacceptable runtimes. We therefore develop the following two methods for speeding up path finding for single trains.

**Safe Interval Path Planing (SIPP)** SIPP (Phillips and Likhachev 2011) is an advanced variant of space-time A* that groups contiguous, collision-free timesteps into safe intervals and represents each state by a pair of a cell and a safe interval. The maximum number of safe intervals for one cell is usually significantly smaller than the number of timesteps, which leads to faster planning and smaller memory requirements. On our server, replacing space-time A* with SIPP reduced the runtime of our software on each instance by up to 4 times, and we therefore solved 6 more instances within 8 hours and improved the accumulated reward by 4.576.

**Lazy Planning** Even after we replaced space-time A* with SIPP, single-train path finding could still be time-consuming when there are thousands of trains. Figure 5(a) shows the runtime of SIPP for each train on a Test_36 Level_0 instance that consists of 3,256 trains on a $229 \times 229$ grid map. SIPP runs extremely fast in the beginning, spending approximately $10^{-5}$ seconds per train. However, as the paths of more trains are planned, the runtime of SIPP grows exponentially because it has to find paths that avoid collisions with larger and larger numbers of paths. In the end, SIPP spends more than 34 seconds to find a single path.

We addressed this issue with *lazy planning* where we plan paths for only some of the trains in the beginning, then let the trains move, and plan paths for the rest of trains during execution. Specifically, for each instance, we run PP for at most 100 seconds and then let the trains start to move. If there remain trains that do not have paths, we plan paths for them at timesteps 500, 600, $\cdots$, each time with a runtime limit of 6 seconds. Figure 5(b) shows the resulting runtime distribution.The runtime of SIPP per train during execution was significantly reduced because we increase the departure times of the trains and SIPP thus needs to avoid collisions with fewer paths. Although increasing the departure times of trains may increase their arrival times, lazy planning has two benefits for large instances with non-zero malfunction rates: (1) it avoids pushing too many trains into the environment at once, which can prevent severe traffic jams, as in the example discussed in Section 4.3, from happening; and (2) when planning paths during execution, we take the influence of the malfunctions that have already happened or are happening into account, so the new paths can result in smaller arrival times eventually.

On our server, lazy planning reduced the runtime of our software on each instance by up to 5 times, and we therefore solved 5 more instances within 8 hours and improved the accumulated reward by 3.282. In particular, on the instances that were solved without lazy planning, lazy planning reduced the runtime by 3,117 seconds in total and decreased the accumulated reward by only 0.018.

### 4.5 Summary

In summary, we use PP with SIPP to generate an initial solution and parallel LNS with leader priority to improve its quality. The maximum numbers of LNS iterations are determined by SA. If PP fails to plan paths for all trains within 200 seconds (although we later determined that 100 seconds would work even better), we use lazy planning to continue planning paths during execution. During execution, we use MCP to generate action commands and partial replanning with SIPP to improve the solution quality when new malfunctions occur. On the leaderboard, we eventually solved 362 instances with a success rate of 98.5% within 8 hours and achieved a score (= accumulated reward) of 297.507, which is 1.160 and 24.168 higher than the scores of the second and third teams, respectively. The largest instance we solved with 100% success rate contains 3,256 trains, and we solved it within 704 seconds (including both planning and simulation time) for a reward of 0.802.

## 5 Deadlocks or Why Reinforcement Learning Failed to Win the Challenge

The 2020 Flatland Challenge was branded as "Multi Agent Reinforcement Learning on Trains," and the prize structure, awarding prizes for the best Reinforcement Learning (RL) approaches as well as the best overall approaches, was clearly meant to encourage RL approaches. Indeed, how to apply RL to MAPF has also recently been studied in the research community (Sartoretti et al. 2019; Ling, Gupta, and Kumar 2020). Nevertheless, optimization approaches have consistently dominated the competition. In Round 1, the best RL approach got a score of -0.611, ranked 7th on the leaderboard and was 6 times worse than our score. In Round 2, the best RL approach spent 8 hours to reach a score of 214.15 and ranked 8th on the leaderboard. Our approach reached

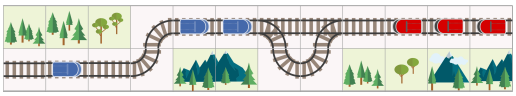


Figure 6: Deadlocks. Although the first two blue trains can pass all red trains (and vice versa), the last blue train will inevitably be blocked.

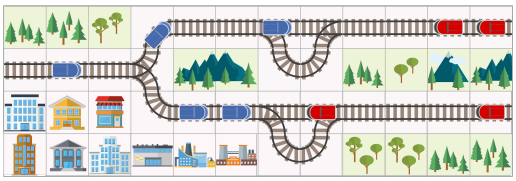


Figure 7: A deadlock arises if the leftmost blue train takes the south rail track.

that score after 15 minutes already. The 2019 Flatland Challenge had similar outcomes.

The reason for these outcomes might be deadlocks. Since trains cannot move backward, any two trains entering a single-track corridor from opposite ends cause a deadlock. Such deadlocks can subsequently block some part of the rail network forever, which makes it harder to route the remaining trains and easier to create further deadlocks, increasing the proportion of the network that is blocked.

Deadlocks are essentially impossible to reason about locally. While local reasoning may suffice for obvious deadlocks, more complicated cases of deadlocks can easily arise when the tracks are densely packed with many trains moving to different target cells, see Figure 6. The only way to prevent deadlocks is to reason globally about the trains. However, even with global observations, RL approaches need to predict future deadlocks, which seems difficult without directly reasoning about paths, as optimization approaches do. Consider the first blue train in Figure 6. It will not be blocked if it enters the south rail track and lets all red trains pass. Similar reasoning applies to the first red train. But these two plans are not incompatible. Similarly, if the leftmost blue train in Figure 7 takes the south rail track at the symmetrical switch, then a deadlock arises. If it takes the north rail track, then all trains can leave the map segment without deadlocks. This issue is difficult to detect from global observations that do not include the paths of all trains. Even worse, as the density of trains grows, the number of situations that can lead to deadlocks increases non-linearly. This issue is, in our opinion, the reason why optimization approaches, which rely on global planners, have consistently outperformed RL approaches across all rounds of both the 2019 and 2020 Flatland Challenges.

## 6 The 2019 Flatland Challenge

The main round (Round 2) of the 2019 Flatland Challenge was similar to Round 1 of the 2020 Flatland Challenge, except that (1) the total number of instances was 250 (instead of 400); (2) the largest instance contained only 250 trains (instead of 400); (3) the runtime limit for generating the next action commands at every timestep was 10 minutes (instead of 5 minutes at timestep 0 and 5 seconds at other timesteps); (4) the objective was to maximize the success rate (instead of the mean normalized reward); and (5) the trains had four different speeds, where a train with "slowness" $k = \{1, 2, 3, 4\}$ can only execute one move action every $k$ timesteps (instead of all trains having "slowness" $k = 1$). Due to these differences, it is hard to compare our approach empirically with the top approaches of the previous year.

While details of the top approaches of the previous year are not readily available, we gather from the presentations of the top three teams made at AMLD 2019[2] that they all used PP to generate initial collision-free paths. To handle malfunctions, the first two teams appear to have used MCP or some similar mechanism with some forms of replanning (which appear to be different from ours since they do not have the 5-second runtime limit per timestep), while the third team appears to have replanned paths for the trains that encounter new malfunctions and the trains that were affected by these malfunctions.

One approach from the 2019 Flatland Challenge that is well described is Complete Path Reservation (CPR) (Wälter 2020). It is an alternative approach to MAPF that runs fast and is guaranteed to be deadlock-free under malfunctions. Its main idea is that, after it plans a path for a train, it reserves the directions of the cells on the path so that future trains cannot traverse these cells in the opposite directions. This rule prevents any two trains from traversing a single-track corridor in opposite directions and thus guarantees deadlock-freeness. The reservations are updated on the fly so that, eventually, it can plan paths for all trains and move all of them to their target cells within a finite number of timesteps (that can be larger than $T_{max}$). However, the solution quality of CPR is dramatically worse than that of our approach. For example, for instances in Test_33 (which contains 1,006 trains), our approach obtained a mean reward of 0.649 with a success rate of 96% on our server, while CPR obtained a mean reward of 0.406 with a success rate of 66%. Due to the low success rate of CPR, it needs more time to run the simulation (as more instances need to be run to timestep $T_{max}$) and thus eventually solved 9 instances fewer than our approach. Its final score was also 44.082 lower than that of our approach. We therefore did not pursue CPR.

## 7 Conclusions

The Flatland Challenge has been a valuable exercise for us. The scalability challenges pushed us to develop extremely fast but reliable algorithms for MAPF planning and replanning. The high-level algorithmic approach that we created for the challenge is very well suited to the real-world problem of train planning and replanning. The combination of prioritized planning and safe interval path planning extends directly to the real-world problem where sectors are of different lengths and the travel times of trains are different for each sector. The use of large neighborhood search and partial replanning run fast and result in high-quality solutions.

[2] https://youtu.be/rGzXsOC7qXg

## Acknowledgments

We would like to thank AIcrowd, Swiss Federal Railways, Deutsche Bahn, and SNCF for organizing the competition. While we did not attempt to generate a reinforcement learning approach to the problem, which is what the competition was looking for, the competition is very valuable to the research community for providing a simplified yet realistic version of the train planning and replanning problem. Finally, we would like to thank and congratulate the second place team MasterFlatland. Without their impressive effort, we would not have worked so hard, and they could just as well have won the competition instead of us.

The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189, 1837779, and 1935712 as well as a gift from Amazon. The Research at Monash University was partially supported by the Australian Research Council (ARC) under grant numbers DP190100013 and DP200100025 as well as a gift from Amazon. The research at Simon Fraser University was supported by the Natural Sciences and Engineering Research Council (NSERC) under grant number RGPIN-2020-06540. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

## References

Atzmon, D.; Stern, R.; Felner, A.; Sturtevant, N. R.; and Koenig, S. 2020. Probabilistic Robust Multi-Agent Path Finding. In *ICAPS*, 29–37.

Cáp, M.; Gregoire, J.; and Frazzoli, E. 2016. Provably safe and deadlock-free execution of multi-robot plans under delaying disturbances. In *IROS*, 5113–5118.

Cohen, L.; Uras, T.; Kumar, T. K. S.; and Koenig, S. 2019. Optimal and Bounded-Suboptimal Multi-Agent Motion Planning. In *SoCS*, 44–51.

Coskun, A.; and O'Kane, J. M. 2019. Online Plan Repair in Multi-robot Coordination with Disturbances. In *ICRA*, 3333–3339.

Ho, F.; Salta, A.; Geraldes, R.; Goncalves, A.; Cavazza, M.; and Prendinger, H. 2019. Multi-Agent Path Finding for UAV Traffic Management. In *AAMAS*, 131–139.

Hönig, W.; Preiss, J. A.; Kumar, T. K. S.; Sukhatme, G. S.; and Ayanian, N. 2018. Trajectory Planning for Quadrotor Swarms. *IEEE Transactions on Robotics* 34(4): 856–869.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2021a. Anytime Multi-Agent Path Finding via Large Neighborhood Search: Extended Abstract. In *AAMAS*, (to appear).

Li, J.; Gange, G.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2020a. New Techniques for Pairwise Symmetry Breaking in Multi-Agent Path Finding. In *ICAPS*, 193–201.

Li, J.; Sun, K.; Ma, H.; Felner, A.; Kumar, T. K. S.; and Koenig, S. 2020b. Moving Agents in Formation in Congested Environments. In *AAMAS*, 726–734.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. K. S.; and Koenig, S. 2021b. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. In *AAAI*, (to appear).

Li, J.; Zhang, H.; Gong, M.; Liang, Z.; Liu, W.; Tong, Z.; Yi, L.; Morris, R.; Pasareanu, C.; and Koenig, S. 2019. Scheduling and Airport Taxiway Path Planning under Uncertainty. In *AIAA Aviation Forum*.

Ling, J.; Gupta, T.; and Kumar, A. 2020. Reinforcement Learning for Zone Based Multiagent Pathfinding under Uncertainty. In *ICAPS*, 551–559.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with Consistent Prioritization for Multi-Agent Path Finding. In *AAAI*, 7643–7650.

Ma, H.; Kumar, T. K. S.; and Koenig, S. 2017. Multi-Agent Path Finding with Delay Probabilities. In *AAAI*, 3605–3612.

Ma, H.; Li, J.; Kumar, T. K. S.; and Koenig, S. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In *AAMAS*, 837–845.

Ma, H.; Wagner, G.; Felner, A.; Li, J.; Kumar, T. K. S.; and Koenig, S. 2018. Multi-Agent Path Finding with Deadlines. In *IJCAI*, 417–423.

Mohanty, S. P.; Nygren, E.; Laurent, F.; Schneider, M.; Scheller, C.; Bhattacharya, N.; Watson, J. D.; Egli, A.; Eichenberger, C.; Baumberger, C.; Vienken, G.; Sturm, I.; Sartoretti, G.; and Spigler, G. 2020. Flatland-RL: Multi-Agent Reinforcement Learning on Trains. *CoRR* abs/2012.05893.

Okoso, A.; Otaki, K.; and Nishi, T. 2019. Multi-Agent Path Finding with Priority for Cooperative Automated Valet Parking. In *ITSC*, 2135–2140.

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe Interval Path Planning for Dynamic Environments. In *ICRA*, 5628–5635.

Ropke, S.; and Pisinger, D. 2006. An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science* 40(4): 455–472.

Sartoretti, G.; Kerr, J.; Shi, Y.; Wagner, G.; Kumar, T. K. S.; Koenig, S.; and Choset, H. 2019. PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning. *IEEE Robotics Automation Letters* 4(3): 2378–2385.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-Based Search for Optimal Multi-Agent Pathfinding. *Artificial Intelligence* 219: 40–66.

Shaw, P. 1998. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *CP*, 417–431.

Sigurdson, D.; Bulitko, V.; Yeoh, W.; Hernández, C.; and Koenig, S. 2018. Multi-Agent Pathfinding with Real-Time Heuristic Search. In *CIG*, 1–8.

Silver, D. 2005. Cooperative Pathfinding. In *AIIDE*, 117–122.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *SoCS*, 151–159.

Street, C.; Lacerda, B.; Mühlig, M.; and Hawes, N. 2020. Multi-Robot Planning Under Uncertainty with Congestion-Aware Models. In *AAMAS*, 1314–1322.

Svancara, J.; Vlk, M.; Stern, R.; Atzmon, D.; and Barták, R. 2019. Online Multi-Agent Pathfinding. In *AAAI*, 7732–7739.

Wagner, G.; and Choset, H. 2017. Path Planning for Multiple Agents under Uncertainty. In *ICAPS*, 577–585.

Wälter, J. 2020. *Existing and Novel Approaches to the Vehicle Rescheduling Problem (VRSP): In the Course of the Flatland Challenge by Swiss Federal Railways (SBB)*. Master's thesis, University of Applied Sciences Rapperswil, Rapperswil, Switzerland.